Mälardalen University Press Dissertations
No. 108

# SOFTWARE ARCHITECTURE EVOLUTION THROUGH EVOLVABILITY ANALYSIS

**Hongyu Pei Breivold**

**2011**


School of Innovation, Design and Engineering

ISBN 978-91-7485-040-6
ISSN 1651-4238
Printed by Mälardalen University, Västerås, Sweden

Mälardalen University Press Dissertations
No. 108

# SOFTWARE ARCHITECTURE EVOLUTION THROUGH EVOLVABILITY ANALYSIS

Hongyu Pei Breivold

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
måndagen den 14 november 2011, 14.15 i Beta, Mälardalens högskola, Västerås.

Fakultetsopponent: Dr Ipek Ozkaya, Carnegie Mellon University


Akademin för innovation, design och teknik

Abstract

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 two main aspects: (i) what software characteristics are necessary for an evolvable software system; and (ii) how to assess evolvability of long-lived proprietary systems in a systematic manner. A secondary focus is to investigate how evolvability is addressed in open source software evolution.

We have performed a systematic review of architecture evolution research, and proposed a software evolvability model, in which subcharacteristics of software evolvability and corresponding measuring attributes are identified. Based on this model, we have proposed the softwarearchitectureevolvabilityanalysis (AREA) process which provides repeatable techniques for supporting software architecture evolution:

a) Qualitative evolvability analysis method that focuses on improving the capability of being able to understand and analyze systematically the impact of change stimuli on software architecture evolution;

b) Quantitative evolvability analysis method that provides quantifications of stakeholders' evolvability concerns and potential architectural solutions' impacts on evolvability.

These techniques have been validated in industrial settings of different domains, and can be used as an integral part of software development and evolution process. This is to ensure that the implications of the potential improvement strategies and evolution path of software architectures are analyzed with respect to the evolvability subcharacteristics.

As a supplementary research contribution, we have conducted a systematic review of the existing studies in open source software (OSS) evolution, and performed a comprehensive analysis which describes how software evolvability is addressed during the development and evolution of OSS, and identified challenges and future research directions in OSS evolution.

ISBN 978-91-7485-040-6
ISSN 1651-4238

*To my parents, Xiaotian and Xincai, who made this all possible through your love, encouragement, and sacrifice.*

*To my love Jon, and my precious Johanna, Martin, and Elin, who have been with me every step of the way and filled my world with strength, courage and meaning.*

*To the memory of Lasse Sletmo, my friend and mentor, who would have been proud.*

# Abstract

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 two main aspects: (i) what software characteristics are necessary for an evolvable software system; and (ii) how to assess evolvability of long-lived proprietary systems in a systematic manner. A secondary focus is to investigate how evolvability is addressed in open source software evolution.

We have performed a systematic review of architecture evolution research, and proposed a software evolvability model, in which subcharacteristics of software evolvability and corresponding measuring attributes are identified. Based on this model, we have proposed the software architecture evolvability analysis (AREA) process which provides repeatable techniques for supporting software architecture evolution:

a) Qualitative evolvability analysis method that focuses on improving the capability of being able to understand and analyze systematically the impact of change stimuli on software architecture evolution;

b) Quantitative evolvability analysis method that provides quantifications of stakeholders' evolvability concerns and potential architectural solutions' impacts on evolvability.

These techniques have been validated in industrial settings of different domains, and can be used as an integral part of software development and evolution process. This is to ensure that the implications of the potential improvement strategies and evolution path of software architectures are analyzed with respect to the evolvability subcharacteristics.

As a supplementary research contribution, we have conducted a systematic review of the existing studies in open source software (OSS) evolution, and performed a comprehensive analysis which describes how software evolvability is addressed during the development and evolution of OSS, and identified challenges and future research directions in OSS evolution.

# Acknowledgements

It is often said that, “all of life is a journey. Which roads we take, what we look back on, and what we look forward to, is up to us.” When I look back over the past five years and the years behind, I cannot believe what an incredible journey it has been. So much has happened, and I feel so very fortunate that I have been able to live this dream. For this, I owe my sincerest gratitude to my old friend, Lasse Sletmo, who introduced me to Sweden in the first place and made this entire incredible journey possible.

Many wonderful people have worked hard, and supported me so much through the past five years. I am grateful to the many dedicated individuals who have supported me for so long, and I would like to thank you all.

My heartfelt thanks go to my main supervisor Prof. Ivica Crnkovic for believing in me, and for making the creation of this dissertation a thoroughly constructive and enjoyable experience. You have been teaching me so much over the past five years, and inspiring me with the example of your hard work and intensity. I am truly grateful for having the opportunity to work under your direction. Thank you for being unfailingly generous with your time, your knowledge and experience, giving me good advice and support when it is needed, and for giving me the opportunities that have helped me in academia.

Many thanks go also to my assistant supervisor Prof. Magnus Larsson for your constant support and encouragement throughout this work, for providing me the great flexibility to combine work and studies, and for sharing your talent and inspiration with me. I am grateful to Prof. Ivica Crnkovic, and Prof. Magnus Larsson, who made time in their very busy schedules, carefully reading through the dissertation in its various stages and giving me practical feedback that have led to great improvements of this work. Very special thanks to Prof. Sasikumar Punnekkat and Dr. Patricia Lago for insightful comments on my initial proposal and licentiate thesis, which paved the way for shaping the dissertation into its ultimate form. Special thanks go also to Dr. Muhammad Ali Babar, Dr. Rikard Land, Dr.

Stig Larsson, Dr. Daniel Sundmark, and Muhammad Aufeef Chauhan for nice cooperation.

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, Dr. Radu Dobrin for advice on presentations, Gunnar Widforss, Harriet Ekwall, Monica Wasell, and Carola Ryttersson 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. Very special thanks to Helena Malmqvist for constant support.

I would also like to acknowledge Swedsoft (http://www.swedsoft.se/) for providing the forum for cooperation and exchange of ideas between companies, and Ericsson for making it possible to conduct our case study. In particular, I would like to thank Anders Fredén, Magnus Norlander, Erik Lind, and all the others involved in the case study at Ericsson for their help and participation in our case study, and for their valuable input and positive feedback.

Over the years, I have had the opportunity to get acquaintance with the best creative people who have given much joy and moral support. I especially want to thank Séverine Sentilles, Aneta Vulgarakis, Juraj Feljan, Cristina Seceleanu, Tiberiu Seceleanu, Adnan Causevic, Aida Causevic, Hüseyin Aysan, Moris Behnam, Yue Lu, Farhang Nemati, Dag Nyström, Stefan Bygde, Leo Hatvani, Jagadish Suryadevara, Yina Zhang, Chen Yang Steen, and Anna Östholm for your friendship. I would also like to thank all my friends from Arosia Network for all the cheerful talks over lunches and nice company in various activities.

This work would not be possible without the support of my family. I especially want to thank my parents, who have sacrificed a good many years of their lives to let me go and pursue my dreams. They have always understood that it is impossible for me to visit them or to be with them as often as they would have liked. Thank you for seeing me through every step of the way, for cheering me up during times of trouble, and for rejoicing with me over every little triumph, despite the distance between us. Many thanks go to my brother for always caring about me and being there for me. I want also to express my immense appreciation to Anita Sletmo, and Stig

Lundvall, who have been one inseparable part of our family throughout the years. Your courage and determination to stand up to various difficult challenges in life have been inspiring me and pushing me forward. Finally, I would like to thank my beloved husband and my wonderful children for sharing my dreams and for all the joy you brought to my life!

Hongyu Pei Breivold

Shenyang, July, 2011

# Contents

# Introduction

It has been recognized that, for long-lived industrial software, the largest part of lifecycle costs is concerned with the evolution of software to meet changing requirements [22]. To keep up with new business opportunities, the need to change software on a constant basis with major enhancements within a short timescale puts critical demands on the software system's capability of rapid modification and enhancement to achieve cost-effective software evolution.

According to Madhavji et al. [119], 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 [185]. It is one term that expresses the software changes during a 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 [130]. In this context, software evolvability is an attribute that is used to describe the software system's capability to accommodate changes. To better explain the term evolvability, we refer to the definition of *Software Evolvability* by Rowe et al. [154]:

*"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 ever-changing world makes evolvability a strong quality requirement for the majority of software architectures [26, 153]. The inability to effectively

and reliably evolve software systems means loss of business opportunities [21].

According to Weiderman et al. [177], software evolvability is a fundamental element for an efficient implementation of strategic decisions and increasing economic value of software. Thus, the need for greater system evolvability is becoming recognized [153]. We have also observed this need from various cases in industrial context [33, 53], 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 software systems in terms of evolvability. Therefore, the motivation of this thesis is to define 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. Describe the existing research studies in architecting for evolvability, and identify important challenges and future research directions in software architecture evolution;
3. Assess software evolvability in a systematic manner, with focus on proprietary systems;
4. Describe how evolvability is addressed in open source software evolution, and identify important challenges and future research directions in open source software evolution.

## 1.2 Research Context

This section explains the scope of research context for this thesis. We focus on software architecture evolution of proprietary systems, the "*how*" perspective of software evolution, and architectural analysis techniques. Moreover, we look into open source software area as a complementary research focus, and analyze how evolvability is addressed in open source software evolution.

### 1.2.1 Proprietary Systems in Focus

The software systems that we have worked with throughout this research are legacy systems that represent valuable software assets. Therefore, the focus

of our research is primarily aimed at analyzing software evolvability for industrial systems that often have a lifetime of 10-30 years and are continuously changing. 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. Software systems must often reflect these changes to adequately fulfill their roles and remain relevant to stakeholders. Therefore, software evolvability was identified in these cases as a very important quality attribute that must be continuously maintained during their lifecycle.

Moreover, these systems most likely have gone through many staff turnovers of the original developers. Thus they show signs of many modifications and adaptations. They also have the typical characteristics of legacy systems as described by Demeyer et al. [60], e.g., increasing complexity, poor documentation, and lack of understanding by the current developers. For such systems, there is a need to address explicitly evolvability during the entire lifecycle in order to prolong their productive lifetime.

### 1.2.2 Open Source Software as Complementary Focus

A complementary research focus is open source software evolution, as the emergence of the open source software paradigm provides researchers with access to the code bases of a large number of evolving software systems along with their release histories and change logs. This has led to an interest in the empirical study of software evolution. Moreover, as some of the open source software projects have become long-lived products, e.g., OSS operating system Linux, some findings in open source have also emerged to compare the evolution of open source and proprietary systems. We can notice that the evolution of OSS becomes as important as for proprietary systems. In this aspect, we assume that, while the reasons of the OSS evolution might be similar as those for proprietary systems, the mechanisms and the characteristics can be different. Therefore, we collected information based on the existing literatures, and performed a comprehensive analysis in assessing and interpreting all available research studies instead of focusing on a particular open source software project. In doing so, we attempt to examine characteristics of evolving open source systems and analyze how evolvability is addressed in open source software evolution.

### 1.2.3 "How" Perspective of Software Evolution in Focus

Lehman [113] describes two perspectives on software evolution: "*what and why*" versus "*how*". The "*what and why*" perspective studies the nature of the software evolution phenomenon, and investigates its driving factors and impacts. In this research, we focus on the "*how*" perspective of software evolution, and address the pragmatic aspects, i.e., the development of methods and tools that provide the means to control software evolution.

### 1.2.4 Software Architecture Evolution in Focus

According to Mens and Demeyer [128], one of the main challenges of software evolution is that all artefacts produced and used during the entire software lifecycle are subject to changes, ranging from early requirements over analysis and design documents, to source code and executable code. Consequently, there are many sub-disciplines within the area of software evolution, e.g., requirement evolution, architecture evolution, implementation evolution. In the meanwhile, analyzing and improving software evolution can be done through various ways, e.g., analyzing release histories, source code, and software architecture level.

Software systems undergo two main kinds of evolution [128], i.e., internal evolution and external evolution. This 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.

Our research focuses on the software architectural evolution for two reasons. Firstly, Bass et al. [18] states that, the foundation for any software system is its architecture, which allows or precludes nearly all of the quality attributes of the system. For instance, a system without an adaptable architecture will degenerate sooner than a system based on an architecture that takes changes into account [71]. Secondly, the architecture of a software system describes its high level structure and behavior. Thus, software architecture exposes the dimensions along which a system is expected to evolve [74], and provides basis for software evolution [126]. Therefore, architecture evolution permits

planning and system restructuring at a high level of abstraction where quality and business tradeoffs can be analyzed [75].

### 1.2.5 Architectural Analysis Techniques in Focus

In this thesis, we focus on architectural aspects, and propose architectural approaches that are concerned with software architecture analysis and software quality improvement related to software evolvability. Nevertheless, software evolution spawns also research disciplines that are devoted to the topic of migrating or reengineering legacy software systems by applying a specific software development paradigm to facilitate software evolution, e.g., product line engineering, component-based software engineering, and service-oriented software engineering. However, due to the variety of software development paradigms and the many sub-disciplines concerned in each paradigm, we have chosen to constrain the scope of the thesis to architectural analysis techniques that help analyze and improve software evolvability. For those who are interested in the specific reengineering techniques that facilitate software architecture evolution, please refer to my licentiate thesis [30], which described the impact analysis of the introduction of service-oriented software engineering to component-based software engineering, as well as managing the migration of legacy systems towards product lines.

## 1.3 Research Questions

We describe in the previous sections 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. In this thesis, the system in focus is industrial software system. Thus, the next question relates to the assessment of software evolvability of this type of system:

*How to assess software evolvability of long-lived proprietary systems in a systematic manner?* **(Q2)**

With the emergence of the open source paradigm, researchers are also provided with a wealth of data for open source software (OSS) evolution analysis. Moreover, as more empirical findings in open source have emerged that appear to diverge from the classical view of proprietary systems, studying OSS evolution is becoming important in order to investigate if the mechanisms and concerns for evolution could be different between open source and proprietary systems. Therefore, as a supplementary research, the next question relates to studying how evolvability is addressed in OSS evolution:

*How is software evolvability addressed in the development and evolution of open source software?* **(Q3)**

## 1.4 Research Contributions

Motivated by the need to understand software architecture evolution and to investigate ways to analyze software evolvability to support this evolution, the central theme of this thesis focuses on four particular aspects:

- Identify software characteristics that are necessary to constitute an evolvable software system;
- Assess evolvability in a systematic manner, with focus on proprietary systems;
- Describe existing research studies in architecting for evolvability, and identify important challenges and future research directions in software architecture evolution;
- Describe existing research studies in open source software evolution, and identify important challenges and future research directions in open source software evolution.

The main contributions of the research include:

- **Software evolvability model**

The software evolvability model refines evolvability into a collection of subcharacteristics that can be measured through a number of measuring attributes, and is established as a first step towards analyzing and quantifying evolvability. This model provides a basis for analyzing software evolvability, and a check point for evolvability evaluation and improvement.

- **Software architecture evolvability analysis (AREA) process**

  The AREA process provides repeatable techniques for supporting software architecture evolution. These techniques are based on the software evolvability model, and have been validated through our participation in two industrial projects of different domains, driven by the need of improving software evolvability. The experiences and lessons learned from applying the qualitative analysis method in an industrial case study provided input to the formulation of the quantitative software evolvability analysis method, which is a further refinement and extension of the qualitative evolvability analysis method. The evolvability analysis techniques include:

  - **Qualitative evolvability analysis method**

    The qualitative analysis method focuses on improving the capability of being able to understand and analyze systematically the impact of change stimuli on software architecture evolution.

  - **Quantitative evolvability analysis method**

    The quantitative analysis method provides quantifications of stakeholders' evolvability concerns and potential architectural solutions' impacts on evolvability.

- **Systematic review of architecting for software evolvability**

The systematic literature review of software architecture evolution research synthesizes the existing studies in analyzing and achieving software evolvability at architectural level. The identified primary studies cover a spectrum of approaches with specific perspective or focus on a particular architecture-centric activity in the software lifecycle. A comprehensive overview and analysis of these studies is described. The implications for research and practitioners are identified as well.

- **Systematic review of open source software evolution**

The systematic literature review of open source software (OSS) evolution research analyzes how software evolvability is addressed during the

development and evolution of OSS. The challenges and future research directions in OSS evolution are identified as well.

To summarize, the contributions of the thesis are visualized in Figure 1-1.

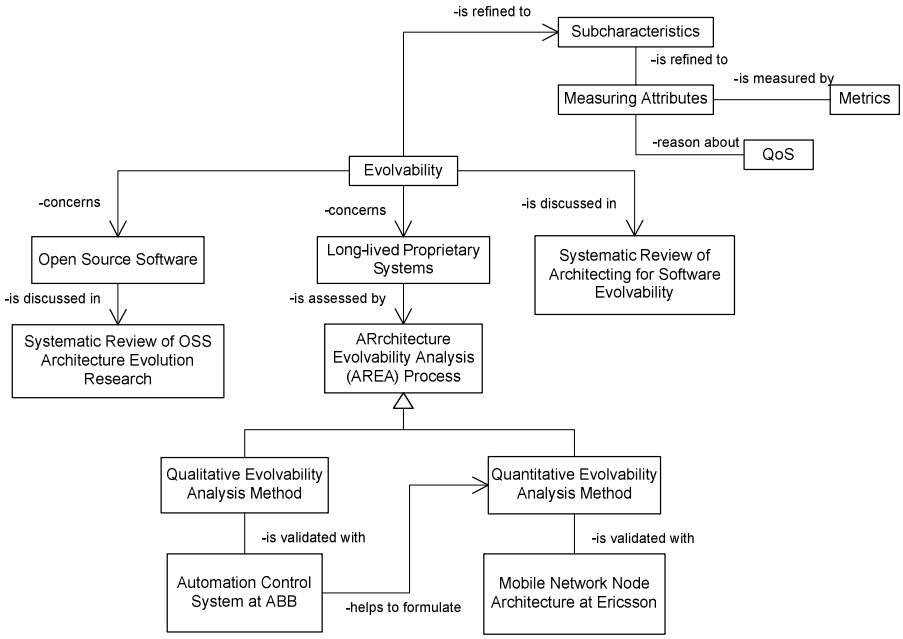


**Figure 1-1: Research contributions of the thesis**

### 1.4.1 Description of Key Publications

The following publications are the basis for the thesis.

**Journals**

- Software Architecture Evolution through Evolvability Analysis, Hongyu Pei Breivold, Ivica Crnkovic, Magnus Larsson, submitted to Elsevier Journal of Systems and Software, 2011.

  **Abstract**: Software evolvability is a multifaceted quality attribute that describes a software system's ability to easily accommodate future changes. It is a fundamental characteristic for an efficient implementation of strategic decisions, and increasing economic value of software. For long-lived systems, there is a need to address evolvability explicitly during the entire software lifecycle in order to

prolong the productive lifetime of software systems. However, designing and evolving a software architecture is a challenging task. This is mainly due to the fact that architecting for evolvable systems implies a complex decision-making process in which multiple aspects need to be taken into consideration, e.g., stakeholders' needs and goals, multiple quality requirements with competing priorities, various architectural solutions with divergent implications on quality requirements. To improve the capability of being able to understand and analyze systematically the evolution of software system architectures, we describe, in this paper, software architecture evolution characterization, and propose an architecture evolvability analysis process that provides repeatable techniques for performing the activities to understand and support software architecture evolution. The activities are embedded in: (i) the application of a software evolvability model; (ii) a structured qualitative method for analyzing evolvability at the architectural level; and (iii) a quantitative evolvability analysis method with explicit and quantitative treatment of stakeholders' evolvability concerns and potential architectural solutions' impacts on evolvability. The qualitative and quantitative assessments manifested in the evolvability analysis process have been validated through their applications in two large-scale industrial software systems at ABB and Ericsson. The experiences and reflections in the case studies with respect to managing software architecture evolution guided by the evolvability analysis at architectural level are described as well in the paper.

**My contribution**: I was the main author, and contributed with the idea and definition of the software evolvability analysis process along with its validation in industrial settings.

**Usage in thesis**: This article is the basis for Chapter 4 and 5 in this thesis, and describes the software evolvability analysis process along with its applications in industrial settings.

- A Systematic Review of Software Architecture Evolution Research, Hongyu Pei Breivold, Ivica Crnkovic, Magnus Larsson, Journal of Information Software and Technology, doi: 10.1016/j.infsof.2011.06.002, 2011.

  **Abstract**: Software evolvability describes a software system's ability to easily accommodate future changes. It is a fundamental characteristic for making strategic decisions, and increasing

economic value of software. For long-lived systems, there is a need to address evolvability explicitly during the entire software lifecycle in order to prolong the productive lifetime of software systems. For this reason, many research studies have been proposed in this area both by researchers and industry practitioners. These studies comprise a spectrum of particular techniques and practices, covering various activities in software lifecycle. However, no systematic review has been conducted previously to provide an extensive overview of software architecture evolvability research. In this work, we present such a systematic review of architecting for software evolvability. The objective of this review is to obtain an overview of the existing approaches in analyzing and improving software evolvability at architectural level, and investigate impacts on research and practice. The identification of the primary studies in this review was based on a pre-defined search strategy and a multi-step selection process. Based on research topics in these studies, we have identified five main categories of themes: (i) techniques supporting quality consideration during software architecture design, (ii) architectural quality evaluation, (iii) economic valuation, (iv) architectural knowledge management, and (v) modeling techniques. A comprehensive overview of these categories and related studies is presented. The findings of this review also reveal suggestions for further research and practice, such as (i) it is necessary to establish a theoretical foundation for software evolution research due to the fact that the expertise in this area is still built on the basis of case studies instead of generalized knowledge; (ii) it is necessary to combine appropriate techniques to address the multifaceted perspectives of software evolvability due to the fact that each technique has its specific focus and context for which it is appropriate in the entire software lifecycle.

**My contribution**: I was the main author, and contributed with leading and conducting the systematic literature review in software architecture evolution research as well as analyzing and synthesizing the results.

**Usage in thesis**: This article is the basis for Chapter 3 in this thesis, and describes a systematic literature review of the software architecture evolution research in architecting for software evolvability.

**Licentiate thesis**

- Software Architecture Evolution and Software Evolvability, Hongyu Pei Breivold, Licentiate Thesis, ISBN 978-91-86135-15-7, Mälardalen University Press, January, 2009.

  **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 of 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.

  **Usage in thesis**: The licentiate thesis is the basis for Chapter 2 in this dissertation, and describes the topic of migrating or reengineering legacy software systems by applying specific software development paradigms, which complement the dissertation.

Examples of specific software development paradigms include component-based software engineering, service-oriented software engineering, and product line software engineering.

**Conferences and workshops**

- A Systematic Review of Studies of Open Source Software Evolution, Hongyu Pei Breivold, Muhammad Aufeef Chauhan, Muhammad Ali Babar, 17th Asia Pacific Software Engineering Conference (APSEC), IEEE, Sydney, Australia, November, 2010.

  **Abstract**: Software evolution relates to how software systems evolve over time. With the emergence of the open source paradigm, researchers are provided with a wealth of data for open source software evolution analysis. In this paper, we present a systematic review of open source software (OSS) evolution. The objective of this review is to obtain an overview of the existing studies in open source software evolution, with the intention of achieving an understanding of how software evolvability (i.e., a software system's ability to easily accommodate changes) is addressed during development and evolution of open source software. The primary studies for this review were identified based on a pre-defined search strategy and a multi-step selection process. Based on their research topics, we have identified four main categories of themes: software trends and patterns, evolution process support, evolvability characteristics addressed in OSS evolution, and examining OSS at software architecture level. A comprehensive overview and synthesis of these categories and related studies is presented as well.

  **My contribution**: I was the main author, and contributed with classification and analysis of the studies included in the systematic literature review.

  **Usage in thesis**: This paper is the basis for Chapter 6 in this thesis, and describes a systematic literature review of the studies in open source software evolution.

- An Extended Quantitative Analysis Approach for Architecting Evolvable Software Systems, Hongyu Pei Breivold, Ivica Crnkovic, Computing Professionals Conference Workshop on Industrial Software Evolution and Maintenance Processes (WISEMP), IEEE, Montréal, Québec, Canada, April, 2010.

  **Abstract**: For long-lived systems, there is a need to address evolvability, i.e., a system's ability to easily accommodate changes,

explicitly during the entire lifecycle. To improve the capability of being able to understand and analyze systematically software architecture evolution, we introduced in our earlier work a software evolvability model and a structured qualitative method for analyzing evolvability at the architectural level. As architecture is influenced by system stakeholders representing different concerns and goals, the business and technical decisions that articulate the architecture tend to exhibit tradeoffs and need to be negotiated and resolved. To avoid intuitive choice of architectural solutions, we propose to extend the qualitative method and strengthen its tradeoff analysis with explicit and quantitative treatment of stakeholders' prioritization of evolvability subcharacteristics and their preferences on design solutions. Finally, an example is used to illustrate the concept and applicability of the proposed approach.

**My contribution**: I was the main author, and contributed with the idea and definition of the proposed quantitative software evolvability analysis method.

**Usage in thesis**: This paper is the basis for Chapter 4 in this thesis, and describes the quantitative evolvability analysis method.

- Analysis of Software Evolvability in Quality Models, Hongyu Pei Breivold, Ivica Crnkovic, 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement (SPPI) Track, IEEE, Patras, Greece, August, 2009.

  **Abstract**: For long-lived systems, there is a need to address evolvability explicitly. For this purpose, we have in our earlier work developed a software evolvability framework based on industrial case studies. With this as input in this paper we analyze several existing quality models for the purpose of evaluating how software evolvability is addressed in these models. The goal of the analysis is to investigate if the elements of the evolvability framework can be systematically managed or integrated into different existing quality models. Our conclusion is that although none of the existing quality models is dedicated to the analysis of software evolvability, we can enrich respective quality model through integrating the missing elements, and adapt each quality model for software evolvability analysis purpose.

**My contribution**: I was the main author, and contributed with the analysis of existing quality models and investigation on how software evolvability is addressed in these quality models.

**Usage in thesis**: This paper is the basis for Chapter 2 in this thesis, and describes how software evolvability is addressed in several existing quality models.

- Analyzing Software Evolvability, Hongyu Pei Breivold, Ivica Crnkovic, Peter Eriksson, 32nd IEEE International Computer Software and Applications Conference (COMPSAC), Turku, Finland, July, 2008.

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

  **My contribution**: I was the main author, and contributed with the proposed evolvability model and the case study in applying the evolvability model.

  **Usage in thesis**: This paper is the basis for Chapter 4 in this thesis, and describes the software evolvability model.

- 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.

  **Abstract**: Evolution of software systems is characterized by inevitable changes of software and increasing software complexity, which in turn may lead to huge maintenance and development costs. For long-lived systems, there is a need to address evolvability (i.e., a system's ability to easily accommodate changes) explicitly in the requirements and early design phases, and maintain it during the

entire lifecycle. This paper describes our work in analyzing and improving the evolvability of an industrial automation control system, and presents 1) evolvability subcharacteristics based on the problems in the case and available literature; 2) a structured method for analyzing evolvability at the architectural level. This paper includes also the main analysis results and our observations during the evolvability analysis process in the case study.

**My contribution**: I was the main author, and contributed with the description of the proposed qualitative software evolvability analysis method, the case study in applying the method, the analysis results and conclusions.

**Usage in thesis**: This paper is the basis for Chapter 4 and 5 in this thesis, and describes the qualitative software evolvability analysis method along with its application in an industrial setting.

### 1.4.2 Other Related Publications

The following publications are related to the thesis.

**Conferences and workshops**

- What Does Research Say About Agile and Architecture?, Hongyu Pei Breivold, Daniel Sundmark, Peter Wallin, Stig Larsson, 5th International Conference on Software Engineering Advances (ICSEA), IEEE, Nice, France, August, 2010.
- A Systematic Review on Architecting for Software Evolvability, Hongyu Pei Breivold, Ivica Crnkovic, 21st Australian Software Engineering Conference (ASWEC), IEEE, Auckland, New Zealand, April, 2010.
- Software Architecture Evolution – An Integrated Approach in Industry (Extended Abstract), Hongyu Pei Breivold, Ivica Crnkovic, 21st Australian Software Engineering Conference (ASWEC), IEEE, Auckland, New Zealand, April, 2010.
- A Systematic Review of Software Evolvability, Hongyu Pei Breivold, Mälardalen University Workshop on Software Engineering, Västerås, Sweden, November, 2009.
- 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), IEEE, L'Aquila, Italy, September, 2008.
- 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.

**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.

**Technical Reports**

- A Survey of Software Architecture Evolvability, Hongyu Pei Breivold, Ivica Crnkovic, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-239/2009-1-SE, Mälardalen Real-Time Research Center, Mälardalen University, September, 2009
- Using Software Evolvability Model for Evolvability Analysis, Hongyu Pei Breivold, Ivica Crnkovic, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-222/2008-1-SE, Mälardalen Real-Time Research Center, Mälardalen University, February, 2008

## 1.5 Research Methodology

The research process and research methods as well as the general validity of the research results are discussed in the following subsections. The detailed validity discussions that are concerned with the research results (e.g., software evolvability model, the qualitative and quantitative evolvability

analysis processes, and systematic review process [100]) in various domains will be described later in Chapter 7.

### 1.5.1 Research Process

The research process conducted in this thesis is illustrated in Figure 1-2.

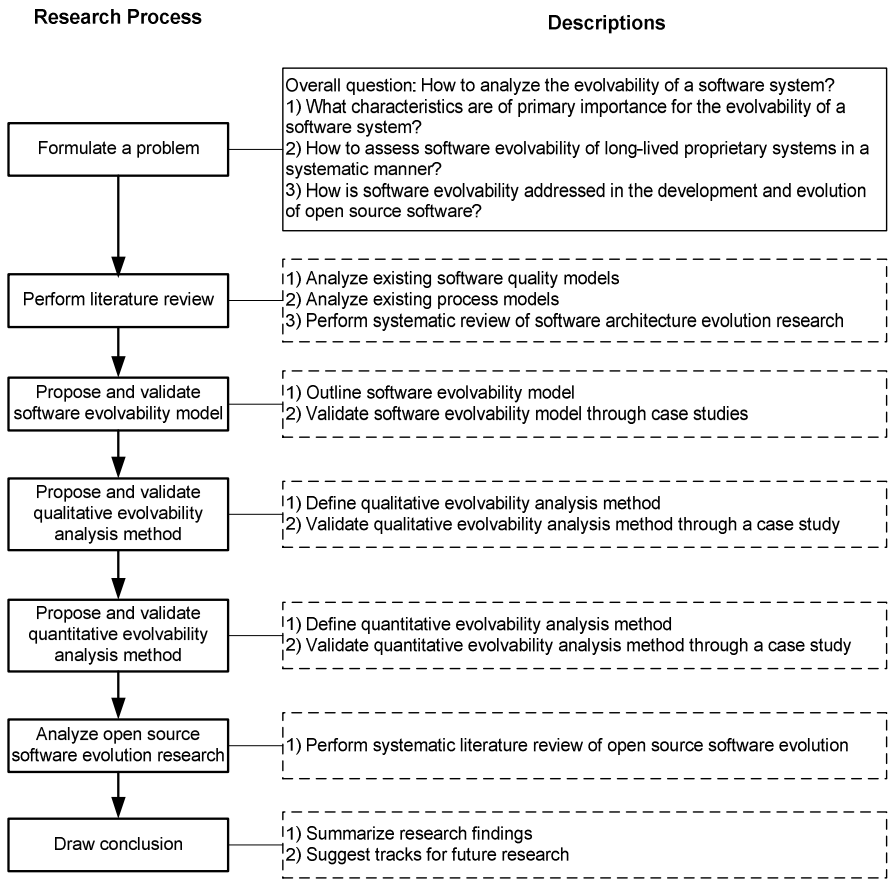


**Figure 1-2: Research process**

The above research phases are not strictly sequential or separated. Firstly, the software evolvability model laid a ground for the subsequent two phases, as both the qualitative and quantitative evolvability analysis methods are based on the software evolvability model. Consequently, the case studies for validating the qualitative and quantitative evolvability analysis methods

provided feedbacks to, and further validated the evolvability model. Secondly, the feedbacks and experiences from the case study for qualitative analysis method provided feedbacks to its refinement, and led to the proposed quantitative analysis method.

The different phases of the research process are explained below:

**Formulate a problem**: The research questions are defined in Chapter 1.3.

**Perform literature review**: We performed a thorough investigation and analysis of the state-of-the-art and state-of-the-practice of the existing well-known software quality models as well as the existing process models for software evolution. In addition, a systematic literature review on architecting for software evolvability was performed with the intention to critically analyze the existing approaches in evaluating and improving software evolvability at architectural level, and to identify implications for practice and future research.

**Propose and validate software evolvability model**: Based on the knowledge from the quality models and process models, the idea of a characterization of software architecture evolution was outlined. In addition, a software evolvability model was created, including also case studies with two development organizations from two different domains to address the issues with software architecture evolution.

**Propose and validate qualitative evolvability analysis method**: The qualitative evolvability analysis method was defined and validated in an industrial setting, and we obtained valuable experiences and feedbacks which were the basis for the formulation of quantitative analysis method.

**Propose and validate quantitative evolvability analysis method**: The quantitative evolvability analysis method was shaped based on the validation results from the qualitative analysis method. Accordingly, a quantitative method complementary to the qualitative evolvability analysis method was proposed. The validation of the quantitative evolvability analysis method was performed in a different industrial domain than for the qualitative analysis method.

**Analyze open source software evolution research**: A systematic literature review was performed with the intention to critically analyze the existing studies in open source software evolution, to describe how software evolvability is addressed during the development and evolution of open source software, and to identify challenges and future research directions in OSS evolution.

**Draw conclusion**: We summarized the findings in our software architecture evolution research with respect to proprietary system and open source software respectively, and discussed how to address their specific challenges in software evolution through evolvability analysis.

### 1.5.2 Research Methods

This section presents an overview of the research methods used in the research presented in this thesis.

A summary of the computing research methods can be found in [87]. Among them, a collection of specific research methods are used in this thesis for data collection, and are classified into two categories: methods related to case study process, and methods related to literature survey process.

*Case Study* [66] is a research technique in which key factors that may affect the outcome of an activity are identified and the activities are documented, including its inputs, constraints, resources and outputs. Two types of case study are described by Yin [183]. They are:

- Single Case: It examines a single organization, group, or system in detail; involves no variable manipulation, experimental design or controls.

  In this research, the idea of the software evolvability model was based on our earlier industrial experiences in working with software systems of different domains. The initial establishment of the evolvability model was validated with a single case.

- 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 Chapter 5 (regarding the application of software evolvability model, and evolvability analysis processes) are derived from two different organizations in two different domains, and belong to the *multiple case studies* category.

From case study process perspective, the following research methods were used for data collection:

- *Interview* [23]: 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. The structured interview has a formalized, limited set of questions, whereas the unstructured interview can pose questions that can be

changed or adapted to meet the interviewee's intelligence and understanding.

In the research presented in this thesis (regarding the case studies in applying the qualitative and quantitative analysis of software evolvability), we performed semi-structured interviews, because we had already defined a framework of themes to be explored, and meanwhile, we wanted to allow new questions to be brought up during the interviews as a result of what the interviewees say.

- *Lessons-learned* [186]: 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.

  Some of the results reported in Chapter 5 (regarding the experiences and lessons learned through the application of the qualitative evolvability analysis process in the first industrial case study) are reflections throughout the case study execution. These reflections were then taken into consideration to further extend the qualitative method with the flexibility in making quantitative evolvability analysis. Thus, the development of the quantitative software evolvability analysis method (as described in Chapter 4) is based on the lessons learned in the first case study. Similarly, the results reported in Chapter 5 (regarding the experiences and lessons learned through the application of the quantitative evolvability analysis process in the second industrial case study) are reflections throughout the second case study execution.

From literature survey process perspective, the following research methods were used for data collection:

- *Critical Analysis of the Literature* [186]: This research method is used to collect and analyze data from published material. Literature search requires the investigator to analyze the results of papers and other documents that are publicly available. Another related research method is *systematic literature review* [100] which is a formalized and repeatable process to document relevant knowledge on a specific subject area for assessing and interpreting all available research related to a research question.

  The research context and background description in Chapter 2 (regarding the analysis of existing software quality models) are originated from the *Critical Analysis of the Literature* method. The

research contents in Chapter 3 (regarding the research studies in architecting for software evolvability) and Chapter 6 (regarding the research studies in open source software evolution) in this thesis are based on the *systematic literature review* method.

Based on the research output we have obtained, there are basically two categories of research methods:

- *Qualitative Research* [76]: 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 Chapter 5 (regarding the stakeholders' views on software evolvability subcharacteristics as well as the impact analysis of potential architectural solutions on evolvability subcharacteristics in the first case study) belong to this category.

- *Quantitative Research* [76]: This method is the collection of numerical data in order to explain, predict and/or control phenomena of interest.

  The results presented in Chapter 5 (regarding the quantification of stakeholders' prioritization and preferences on evolvability subcharacteristics, as well as the quantitative impact analysis of potential architectural solutions on evolvability subcharacteristics in the second case study) belong to this category.

### 1.5.3 Validity

Based on Yin [183], four types of validity are considered: construct validity, internal validity, external validity, and reliability. In general, our software architecture evolution research in this thesis is based on empirical studies. As the ways for the data collection and research design vary for each research result we achieved, we will present detailed validity discussions in Chapter 7, in which we go through each research result and describe respective type of the validation used. Below is a brief summary of the four types of validity along with a short description of how our research results were validated.

- *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 [183]:

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

In this thesis, the systematic reviews of architecting for software evolvability and open source software evolution were validated by using multiple literature databases as sources of information, as well as well-specified research protocols.

- *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*" [183]. It can be increased through the following tactics:
  - Do pattern-matching;
  - Do explanation-building;
  - Address rival explanations;
  - Use logic models.

  In this thesis, the systematic reviews of architecting for software evolvability and open source software evolution were validated based on thorough selection process which comprised of multiple stages to retrieve relevant quality papers. The AREA process faces some threats to internal validity, such as different valuation of evolvability subcharacteristics due to different previous working experiences. More details on this and how it was handled are described in Chapter 7.
- *External validity* concerns the possibilities to generalize the results from a study. It can be increased through the following tactics [183]:
  - Use theory in single-case studies;
  - Use replication logic in multiple-case studies.

  In this thesis, the AREA process was validated in two case studies in two different domains. There was no threat in the selection of participants, and the evolvability analysis methods seemed to be generally applicable. However, one threat to external validity is that there are some similarities between the two cases in terms of

properties of software systems (e.g., large, complex, long-lived, and software-intensive) as well as culture perspective.

- *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 [183]:
  - Use case study protocol;
  - Develop case study database.

  In this thesis, the AREA process consists of repeatable techniques that comprise of well-defined phases and steps for conducting software evolvability analysis. Thus, any researcher can repeat the same research procedure. The systematic reviews have detailed research protocols that describe the search keywords, inclusion and exclusion criteria, as well as databases for retrieving information. It is therefore also repeatable for other researchers to perform the same procedure to reach similar conclusions.

## 1.6 Thesis Overview

The thesis consists of the following chapters:

**Chapter 1 – Introduction** describes the background and motivation to the research, including problem statement and research questions. A general discussion on research methodology is also included, along with a more thorough description of the specific methods used for the different parts of the research.

**Chapter 2 – Software Architecture and Evolution** presents relevant fields of research and practice in software architecture and its evolution.

**Chapter 3 – Architecting for Software Evolvability** presents the results from a systematic literature review in software architecture evolution research. The objective of this chapter is to analyze important research themes in software architecture evolution, especially in analyzing and improving software evolvability at architectural level. Some of the most important challenges and future research directions in software architecture evolution are presented as well.

**Chapter 4 – Analyzing Software Evolvability** describes the software architecture evolution characterization, and proposes a software architecture evolvability analysis (AREA) process that provides repeatable techniques for performing the activities to understand and support software architecture

evolution. The activities are embedded in: (i) the definition of a software evolvability model; (ii) a structured qualitative method for analyzing evolvability at the architectural level; and (iii) a quantitative evolvability analysis method with explicit and quantitative treatment of stakeholders' evolvability concerns and potential architectural solutions' impacts on evolvability.

**Chapter 5 – Analyzing Proprietary Systems** describes the industrial case studies at ABB and Ericsson where the software evolvability model, the qualitative and quantitative software evolvability analysis methods were applied.

**Chapter 6 – Open Source Software Evolution** presents the results from a systematic literature review of open source software (OSS) evolution. The objective of this chapter is to describe an overview of the existing studies in open source software evolution, and to analyze how software evolvability is addressed during the development and evolution of open source software. Some of the most important challenges and future research directions in open source software evolution are presented as well.

**Chapter 7 – Validity Discussions** discusses in details the validity aspects of the research results.

**Chapter 8 – Conclusions and Future Work** concludes the thesis, and outlines future work that formulates potential tracks for future studies.

**Appendix A – Primary Studies in Chapter 3** lists the primary studies that were included in the systematic literature review (SLR) in software architecture evolution research, which is reported in Chapter 3.

**Appendix B – Primary Studies in Chapter 6** lists the primary studies that were included in the systematic literature review (SLR) in open source software evolution research, which is reported in Chapter 6.

# Chapter 2. Software Architecture and Evolution

Software evolution is characterized by inevitable changes of software and increasing software complexities. Some of the observed properties of large software systems noted by Brooks [38] further confirm this:

- *Complexity* is 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.
- *Changeability* The software entity is constantly subject to pressures for change.
- *Invisibility* In software, there is no geometric representation. Instead, there are several distinct but interacting graphs of links that represent different aspects of the system. The invisibility in terms of software structure representation reflects the fact that large amount of tacit architectural knowledge and design decisions are not explicitly represented in the architecture. Consequently, during the evolution of a system, designers can easily violate design rules and constraints arising from design decisions taken previously, leading to architectural drifts and erosion [139] that jeopardizes software evolvability.

All this exhibits the intensified need in having evolvable software systems that accommodate changes in a cost-effective way while maintaining the architectural integrity. Having long-lived proprietary systems in focus, it is

therefore of particular interest in this thesis to seek active measures to ensure the long-term success of software architectures so that the quality of a software system will not gradually degrade as the system evolves. For such long-lived systems, software evolvability needs to be explicitly addressed during the entire lifecycle in order to prolong the productive lifetime of software systems. In line with this, there are research and practice areas to which we relate the work in this thesis:

Chapter 2.1 presents the observed behavior of software systems and challenges of software aging to motivate the thesis.

Software architecture plays the central role in this dissertation to address software evolution challenge, because a software architectures has the potential to provide a foundation for managing software evolution, and is inevitably subject to evolution as well. Therefore, we discuss software architecture evolution in Chapter 2.2.

Recognizing that there are several quality models, which have software quality in focus, we discuss these quality models, and analyze how evolvability is addressed in these models in Chapter 2.3.

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. Moreover, a software process model represents activities and practices that embody strategies for accomplishing software evolution. Among the existing process models, Chapter 2.4 focuses on staged model [21], and discusses the idea of software architecture evolution assessment process.

A topic closely related to our research is concerned with migrating or reengineering legacy software systems by applying a specific software development paradigm or technique to facilitate software evolution. Chapter 2.5 presents briefly an overview of these techniques, and discusses how they are related to evolvability. The techniques include component-based software engineering, service-oriented software engineering, product line engineering, aspect-oriented software development, and model-driven engineering.

## 2.1 Software Evolution

This section presents a brief overview of the observed behavior of software systems and challenges encountered during software evolution.

### 2.1.1 Laws of Software Evolution

The laws of software evolution is formulated by Lehman et al. [111, 114], 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 laws concerning continuing change, increasing complexity, continuing growth, and declining quality are of particular interest for this thesis. In order to keep the system as useful as it was, we must continually develop new features, improve its quality, and adapt it to the ever-changing requirements. Changes imply increasing complexity, which poses a difficult problem, i.e., a successful system needs to be evolved in order to stay successful, but while being evolved, it typically deteriorates and becomes increasingly difficult for humans to understand and modify further unless

this is proactively managed [173]. All these motivate the reasons for this thesis, i.e., when evolving a system, it is a viable strategy to seek methods for systematically analyzing the potential impacts of a change on software evolvability and software architecture evolution. We describe this in Chapter 4 and Chapter 5 of this thesis.

### 2.1.2 Software Aging

Software aging is inevitable. Parnas [137] states that, "*Software, like people, gets old. We can't prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable*."

Parnas uses the metaphor of decay to describe how and why software becomes increasingly brittle over time [137]. 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 [137]:

- 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.

A challenge with evolution is that software systems suffer from software aging while they are adapted to changing requirements due to e.g., architectural erosion, or architectural drift. *Architectural erosion* is defined by Perry and Wolf [139] as "*violations in the architecture that lead to increased system problems and brittleness*". In [139], *architectural drift* is defined as "*a lack of coherence and clarity of form which may lead to architectural violation and increased inadaptability of architecture*". Causes for software aging are, for instance, poor design decisions and changes that damage the architecture, or the lack of conformance between implementation and intended architecture.

The proprietary systems in the focus of this thesis are often based on existing legacy implementations; as legacy systems represent substantial corporate knowledge and investment. These legacy systems are usually critical to the business in which they operate. Therefore, they have been maintained and

evolved to fit existing and expanding markets and customer needs. However, any individual software system will eventually reach an old age when it is no longer cost-effective to modify it. It is therefore of particular interest in this thesis to extend the evolution stage that allows for any kind of modification to the software while remaining architectural integrity preserved, and we suggest guiding software architecture evolution through evolvability analysis.

## 2.2 Software Architecture Evolution

The IEEE 1471-2000 standard [88] definition for *software architecture* is "The *fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution*". Software architectures are created and evolved in a complex environment. The architecture business cycle proposed by Bass et al. [18] defines different factors which influence a software architecture, i.e., stakeholders, developing organization, technical environment, and architect's experience. According to Jansen [93], a software architecture is used for the following purposes:

- *Blue-print* outlines a design for the software of a system.
- *Roadmap* allows planning ahead the evolution of the software of a system, and supports a software architect to align the software with a company's long-term business strategy.
- *Communication vehicle* enables different stakeholders to communicate about the major decisions in order to steer and influence the software of a system.
- *Quality predictor* provides an early indicator of the quality of a software system.

Software architectures have the potential to provide a foundation for managing software evolution, as there are correlations between the described purposes of software architecture and software evolution. From *blue-print* perspective, a software architecture models the structure and behavior of a system. It is therefore the basis of the design process, and a guide for the software development process. In the meanwhile, an architecture needs to be evolved in response to changing requirements of diverse stakeholders. Therefore, an architecture cannot be viewed as simply a description of a static software structure, but as a *roadmap* describing its potential evolution paths. From *communication vehicle* perspective, stakeholders usually come

from different backgrounds, and have different or conflicting concerns that the architecture must address. If architectural decisions are not shared among the stakeholders, it would be difficult to resolve conflicts and set common goals among them, and to agree upon principles and decisions that determine the system's development and its evolution, and thus, resulting in high evolution costs. From *quality predictor* perspective, software architectures can be analyzed, which makes it possible to evaluate alternative architectures before a system is built or an evolution path is chosen. Thus, the architecture evolution permits planning and system restructuring at a high level of abstraction where quality and business tradeoffs can be analyzed.

The *quality predictor* purpose of software architectures brings also a strong motivation for software architecture analysis to assess software evolution. As stated by Clements et al. [55], the foundation of any software system is its architecture, which allows or precludes nearly all of the quality attributes of the system. Therefore, software architectures provide a basis for explicitly documenting quality concerns in order to cope with the challenges in constructing and evolving software systems. Accordingly, apart from the analysis results in terms of specific quality concerns in focus, software architecture analysis serves as frameworks for comparing and identifying the strengths and weaknesses in different architecture alternatives, identifying potential architectural drift and erosion, as well as understanding the underlying architectural tradeoffs during software evolution

A software architecture models the structure and behavior of a system; and presents a high level view of a system, including the software elements and the relationships between them. A software architecture is inevitably subject to evolution. It exposes the dimensions along which a system is expected to evolve [74], and provides basis for software evolution [126]. There exist several approaches in describing and evolving software architecture. Aoyama [6] 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. It was noticed that discontinuous evolution emerges between certain periods of successive continuous evolution.

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 [21]. 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. [17] 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 [92], an architectural design decision is a key concept in software architecture evolution. Capturing design decisions is therefore essential to address architectural knowledge [109] vaporization 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 [174].

Lung et al. [116] describe 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 as well as a set of architectural views for reengineering, analyzing, and comparing software architectures. This framework is used to model several types of information:

- *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 are 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.

A detailed study on the software architecture evolution area is described in Chapter 3.

## 2.3 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 [135]. In quality models, quality attributes are decomposed into various factors, leading to various quality factor hierarchies. In the following subsections, we provide a brief survey of the well-known software quality models, which form the basis for the establishment of our software evolvability model (to be described in Chapter 4), as well as an analysis of how evolvability is addressed in these models.

### 2.3.1 Overview of Quality Models

Some well-known quality models are McCall's quality model [125], Dromey's quality model [62], Boehm's quality model [25], ISO 9126 [89] and FURPS quality model [83].

**McCall's quality model** [125] 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 (i.e., protection of the program from unauthorized access), 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 refines the above three perspectives into a hierarchy of factors, criteria and metrics.

**Boehm's quality model** [25] 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* which describes the ability of a product to transit into another hardware-software environment.
- *Utility* which is further refined into reliability, efficiency and human engineering.
- *Maintainability* which is further refined into testability, understandability (i.e., the purpose of the code is clear to the inspector), and modifiability (i.e., the code facilitates the incorporation of changes, once the nature of the desired change has been determined).

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.

**FURPS quality model** [83] takes into consideration the following characteristics:

- *Functionality* which includes feature sets, capabilities and security.
- *Usability* which includes human factors, consistency in the user interface, online and context-sensitive help, wizards, user documentation, and training materials.
- *Reliability* which includes frequency and severity of failure, recoverability, predictability, accuracy, and mean time between failures (MTBF).
- *Performance* which prescribes conditions on functional requirements such as speed, efficiency, availability, accuracy, throughput, response time, recovery time, and resource usage.
- *Supportability* which includes testability, extensibility, adaptability, maintainability, compatibility, configurability, serviceability, install-ability, and localizability/internationalization.

Two steps are considered in this model: setting priorities and defining quality attributes that can be measured. According to Ortega et al. [135], one disadvantage of this model is that it fails to take into account software portability.

**ISO 9126 quality model** [89] 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.

**Dromey quality model** [62] 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, i.e., *requirement quality*

*model*, *design quality model* and *implementation quality model*. The *design quality model* takes into account explicitly the early stages (analysis and design) of the development process. The focus of the *design quality model* is that a design must accurately satisfy the requirements and be *understandable*, *adaptable* in terms of supporting changes, and is developed using a mature process.

The high-level product properties for the implementation quality model include:

- *Correctness* evaluates if some basic principles are violated, with functionality and reliability as software quality attributes.
- *Internal* measures how well a component has been deployed according to its intended use, with maintainability, efficiency and reliability as software quality attributes.
- *Contextual* deals with the external influences on the use of a component, with software quality attributes in maintainability, reusability, portability and reliability.
- *Descriptive* measures the descriptiveness of a component, with software quality attributes in maintainability, reusability, portability and usability.

The information extracted from each model can be used to build, compare and evaluate the quality of a software product. In this model, characteristics with regard to process maturity and reusability are more explicit in comparison with the other quality models. According to Rawashdeh and Matalkah [146], one disadvantage of the Dromey model is associated with reliability and maintainability, as it is not feasible to judge them before the software system is actually operational in the production area.

### 2.3.2 Analysis of Software Evolvability in Quality Models

The quality characteristics that are addressed in the above quality models are summarized in Table 2-1, which provides useful inputs to our idea about evolvability subcharacteristics.

**Table 2-1: 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 |

As shown in Table 2-1, although software evolvability is one of the most important quality attributes or characteristics of software, 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 software evolvability by Rowe et al. [154], the multi-faceted quality attribute evolvability covers more aspects than adaptability, extensibility, or maintainability. As maintainability is covered in most of the well-known quality models and it is generally considered as most related to evolvability, we study the definitions of maintainability in various quality models, as summarized in Table 2-2.

**Table 2-2: Definitions of maintainability in quality models**

| Quality Models | Maintainability Definition | Focus |
|---|---|---|
| **McCall** | The effort required to locate and fix a fault in the program within its operating environment | Corrective maintenance |
| **Boehm** | It is concerned with how easy it is to understand, modify and test. | Understandability, modifiability and testability |
| **FURPS** | Implicit | Adaptability, extensibility |
| **ISO 9126** | The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. | Analyzability, changeability, stability, testability |

In this dissertation, we distinguish evolvability from maintainability, because they both exhibit their own specific focus, as summarized in Table 2-3. Considering these differences, we have found out that only having a collection of the subcharacteristics of maintainability as defined in the ISO software quality standard [89] is not sufficient to characterize software evolvability (i.e., a system's ability to easily accommodate changes) . This poses one of the goals of 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 software evolvability.

**Table 2-3: Comparisons between evolvability and maintainability**

| Characteristics | Evolvability | Maintainability |
|---|---|---|
| **Software Change Stimuli** | Business model, business objectives, functional and quality requirement, environment, underlying and emerging technologies, new standards, new versions of infrastructure | Defects, functional requirement, revised requirements from customers |
| **Type of Change** | Coarse-grained, long term, higher level, radical functional or structural enhancements or adaptations [177] | Fine-grained, short term, localized change [177] |
| **Focus Activity** | Cope with changes | Keep the system perform functions |
| **Software Structure** | Structural change | Relatively constant |
| **Analysis Scenarios** | Growth scenarios (change scenarios) | Existing use case scenarios |
| **Development Process** | May require corresponding process changes | Relatively constant |
| **Architecture Integrity** | Conformance is required | Conformance is preserved |

## 2.4 Software Process Models

The primary functions of a software process model are to determine the stages involved in software development and evolution, and to establish the transition criteria for progressing from one stage to the next [24]. A software process model represents activities and practices that embody strategies for accomplishing software evolution. 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. Some examples are the waterfall model [155], change mini-cycle process model [182], evolutionary development model [78, 79], spiral model [24], Agile software development [57, 121], and staged model [21].

Our research is in line with the idea in staged model [21], which explicitly takes into account the issue of software aging [137], and 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. In this thesis, we focus on seeking viable method to extend the evolution stage.

Software evolution represents the cycle of activities involved in the development, use, and maintenance of software systems. From inception, a software system goes through initial development, productive operation, and retirement from one generation to another. Accordingly, software architecture evolution is inseparably bound to a process context. Scacchi [158] states that, one activity that is critical to the overall evolution of software systems is architecture evaluation, which helps improve the quality of the software systems being evolved and identify potential opportunities and impacts of upcoming changes. In this thesis, we suggest software architecture evolution assessment processes (see Chapter 4 and Chapter 5) that can be performed at many points during a system's life cycle, e.g., during the design phase to evaluate prospective candidate designs, validating the architecture before further commencement of development, or evaluating architecture of a legacy system that is undergoing modification, extension, or other significant upgrades. It can be used to prompt stakeholders to systematically analyze potential impacts of a change on evolvability so as to avoid an ad hoc architecture evolution. The proposed software architecture evolution assessment process focuses on existing software, and engages stakeholders to examine the emerging changes, to discover the driving architectural requirements, stakeholders' evolvability concerns, and potential architectural solutions' impact on evolvability of a software system. The architecture evolution assessment process is stakeholder focused; it is therefore dependent on the participation of involved stakeholders of various

roles, such as architects, development team, research team, project leader, and product managers.

## 2.5 Techniques and Methods Facilitating Architecture Evolution

This thesis focuses mainly on architectural aspects concerned with software architecture analysis and software quality improvement related to software evolvability. Therefore, the topic of migrating or reengineering legacy software systems by applying a specific software development paradigm or technique to facilitate software evolution is not within the scope of this thesis. However, as it is a topic closely related to our research, we present briefly, in the following subsections, an overview of the studies in the techniques that facilitate software architecture evolution along with a brief summary of how respective technologies are related to evolvability. The techniques include component-based software engineering, service-oriented software engineering, product line engineering, aspect-oriented software development, and model-driven engineering. Detailed descriptions of the techniques and case studies that are related to product line engineering, component-based and service-oriented software engineering are presented in my licentiate thesis [30], and are therefore not in focus of this dissertation. Nevertheless, the AREA process (see Chapter 4) proposed in this dissertation is not constrained by any specific techniques. On the contrary, with frequent advances in software engineering, the first phase of the AREA process ensures a thorough analysis of the impact on software architecture evolvability when introducing any new technologies. This impact analysis phase applies to all techniques to be introduced.

### 2.5.1 Component-Based and Service-Oriented 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 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 [105]. Therefore, the ability to combine the strengths of CBSE and SOSE, and use them in a complementary manner becomes essential. Some research has been done in combining the strengths of CBSE and SOSE for improved quality attributes of software solutions. Jian and Willey [94] propose 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 [175] propose 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 [49] introduce service-oriented concepts into component models to provide support for late binding and dynamic component availability in the component models. O'Brien et al. [133] explore how service-oriented architecture impacts a number of quality attributes, and identify issues and tradeoffs related to these quality attributes. The investigated quality attributes are interoperability, performance, security, reliability, availability, modifiability, testability, usability and scalability.

From evolvability perspective, according to Breivold and Larsson [36], CBSE supports a variety of encapsulation types, ranging from white box exposing all the implementation, or gray box exposing parts of component implementation to black box. In the case of white box and gray box, the component clients have the flexibility to make modifications to the components in order to meet specific needs in their solutions. According to the same study, SOSE provides the feasibility for services to be implemented in diverse technologies and for multiple applications running on different platforms to communicate with each other.

### 2.5.2 Software Product Line Methods

A software product line is defined by Clements and Northrop [56] 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*". According to Pohl et al. [141], 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. It has become one of the most established strategies for achieving large-scale software reuse [63].

Within the area of software product line evolution, Bosch [28] proposes methods for designing software architecture, in particular product line architecture. Two key principles behind software product line engineering are elaborated by Pohl et al. [141]:

- Separation of software development in domain and application engineering;
- Explicit definition and management of variability of the product line across all development artifacts.

Van der Linden et al. [172] describe a four-dimensional software product family engineering evaluation model to determine the status of software family engineering, concerning business, architecture, organization and process.

Faust and Verhoef [65] present metrics for genericity relayering, and migrates multiple instances of a single information system to a product line. Bayer et al. [19] propose the RE_MODEL method to integrate reengineering and product line activities in order 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. Schmid et al. [160] propose the PuLSE™ method to address the different phases of product line development, 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, Stoermer and O'Brien [166] propose MAP (Mining Architectures for Product Lines), which focuses on the feasibility evaluation process of the organization's decision to move towards a product line. Options Analysis for Reengineering [163] is another method for mining existing components for a product line. Maccari and Riva [118] propose to combine reference

architecture and configuration architecture in order 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) [95] and Feature-Oriented Reuse Method (FORM) [96], which use feature models to organize system features into trees of nodes that represent the commonality and variability within a software product line. Another notation is the orthogonal variability model [14, 141], which is a graph of variation points and variants.

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. [61], product line evolution occurs in two dimensions because 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.

According to Pohl et al. [141], 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* elicits 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, and 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.

From evolvability perspective, Kolb et al. [103] state that, having pre-determined variation points as introduced in product line engineering makes it relatively easy to introduce changes during software evolution. This is because variation points help to keep the impact of changes small by enforcing separation of concerns among variants. On the other hand, we need to consider the impact with respect to the software system's behavior, quality and any possible tradeoffs when we introduce any variation point and realization mechanism. For instance, according to Coplien [59], the choice of binding mechanisms and binding time has consequences for flexibility and other concerns.

### 2.5.3 Aspect-Oriented Software Development

Aspect-oriented software development (AOSD) aims to offer an added layer of abstraction that can modularize system-level concerns [128], which are usually crosscutting as they cut across the dominant decomposition of the software. According to Mens and Demeyer [128], these crosscutting

concerns are believed to have negative impact on software quality such as evolvability, maintainability and understandability because understanding and changing crosscutting concerns requires touching many different places in the source code.

Brichau et al. [37] state that, AOSD techniques offer abstraction, modularity, and composition support to reason about crosscutting concerns throughout the software life cycle, i.e., from requirements engineering to architecture and detailed design to implementation, and evolution.

From requirement perspective, crosscutting concerns manifest themselves during requirement engineering [144]. The Early Aspect Mining Tool [157] supports identifying aspects across various requirement documents and searching for known candidates for aspects. After identifying the requirements-level aspects, an XML-based composition language [145] is used to represent and specify the requirements-level aspects' impact on other requirements in the system.

From architecture design perspective, aspect-oriented software architecture includes the explicit definition of aspectual components (or architectural aspects) for modularizing crosscutting concerns at the architectural level [108]. The representation of an aspect-oriented architecture involves the explicit representation of the relations and connectors between the architectural components, as well as the specification of normal and crosscutting interfaces [51], which specify when and how an architectural aspect affects other architectural components. In order to trace the aspectual components to their detailed design and implementation, Chavez et al. [51] propose a modeling language that supports the specification of internal elements of design aspects such as internal methods and attributes. To assist the evaluation of the aspect-oriented design, Garcia et al. [73] propose a framework for assessing reusability and maintainability of aspect-oriented design. Studies [72] and [140] integrate the principles of AOSD into architecture description languages.

From implementation perspective, a concern at implementation level is usually considered as a particular behavior or functionality in a program [3]. A concern's implementation can be scattered over various system modules, or a particular module's implementation is tangled with different concerns. To cope with these crosscutting concerns, aspect-oriented programming (AOP) has emerged to localize a concern's implementation in order to improve modularity, understandability, maintainability and evolvability of the code. According to Mens and Demeyer [128], there are three phases

involved when adopting aspect-oriented programming from software evolution perspective:

- *Aspect exploration* is the activity of identifying and analyzing the crosscutting concerns in a non aspect-oriented system, such as what the crosscutting concerns are, where and how they are implemented, and what their impact on the software quality is.
- *Aspect extraction* is the activity of separating the crosscutting concern code from the original code.
- *Aspect evolution* concerns the evolution of aspect-oriented software.

From evolvability perspective, Mens and Tourwé [127] state that the notion of aspects allow a developer to localize a concern's implementation, and thus improve modularity, understandability, maintainability and evolvability of code. Some studies explore the relation between crosscutting concerns and software quality. For instance, Kulesza [107] computed metrics for both object-oriented and aspect-oriented versions of a medium-scale software system, and observed that the aspect-oriented versions resulted in fewer lines of code, improved separation of concerns, weaker coupling and lower intra-component complexity. However, the study also indicated an increased number of operations and components in the aspect-oriented version as well as a lower cohesion for the aspect-oriented components. Gibbs et al. [77] conducted a case study to compare the maintainability and evolvability of a version of a software system that was restructured with traditional abstraction mechanisms against a version of the same system which was restructured by means of aspects. They found that the aspect-oriented version performed either better or not worse than the non aspect-oriented version when dealing with changes.

### 2.5.4 Model-Driven Development

Model-driven development (MDD) encompasses the use of models and model technologies to increase the level of abstraction of the software development process. As a result, MDD is seen as a way to handle the growing complexity of software development as the development process becomes formal enough to be automated. Thus MDD positively influences software maintenance and evolution.

Some studies focus on MDD in large scale, industrial projects, and describe processes in which legacy systems are reverse engineered to model-driven architecture (MDA). For instance, Mansurov and Campara [120] argue that a

first step in the migration towards MDA is the introduction of modeling in the software development process. They propose an approach to raise the maturity of software architectures to a level where software maintenance and evolution are driven by the architecture instead of by the code. Anda and Hansen [5] conduct a case study to investigate the ease of constructing, the use and the utility of use cases, sequence diagrams and class diagrams in modeling and enhancing legacy software compared with development from scratch. The case was a large development project applying UML in the development of a new version of existing systems, with most of the software being embedded. Boronat [27] presents a framework for automatic legacy system migration in MDA, using rewriting logic as their transformation engine. Reus et al. [148] report a feasibility study in reengineering legacy systems towards a model-driven architecture. Fleurey et al. [69] introduce a model-driven process, which describes software migration in large industrial context. The process includes automatic analysis of the existing code, reverse engineering of abstract high-level models, model transformation to target platform models and code generation.

Some studies focus on the implementation of MDD techniques in software engineering processes. Raistrick [142] describes how MDA and UML are used to model new software functionality in the form of an executable UML model and specify the capabilities of existing key components. Staron [165] examines the factors determining the decision upon adoption of MDD principles as well as the conditions that should be fulfilled in order to increase the chances of succeeding in adopting MDD. Baker et al. [15] describe the industrial experiences in creating rigorous models throughout the development process, thereby enabling the introduction of automation.

From evolvability perspective, some studies focus on quantification and baseline of productivity and quality in industrial MDD projects. Shirtz et al. [161] describe the process of adopting MDD from inception to successful maturation. Based on the industrial experiences in adopting model-driven engineering, it was demonstrated by Weigert et al. [178] that model-driven engineering significantly improves the development process for embedded and distributed systems. In the same study, it was experienced that model-driven engineering has dramatically increased both the quality and the reliability of software developed in the organization, as well as the productivity of systems and software engineers.

### 2.5.5 Reverse Engineering and Reengineering

Reverse engineering [52] 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. According to Arnold [10], reverse engineering is a key to software reengineering because it ensures recovering 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 [52], software restructuring [9] 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*" [181].
- *Forward engineering* implements and builds a software system from the restructured model.

This reengineering process is captured in the horseshoe process model for reengineering [98], which consists of three related processes:

- Code and architecture recovery, and conformance evaluation;
- Architecture transformation;
- Architecture-based development in which the new architecture is instantiated.

From evolvability perspective, reverse engineering helps to understand the architecture or behavior of a large software system when the source code is the main information. One approach that assists in software reengineering is refactoring [70], 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, to make software easier to understand, and help to find bugs [70]. As stated by Opdyke [134], while a refactoring does not change the behavior of a program, it can support software design and evolution by restructuring a program in a way that allows other changes to be made more easily.

## 2.6 Summary

In this chapter, we have provided an overview of relevant research areas to ensure a good understanding of the research context of the thesis.

The software evolution retraces motivate the reasons for the thesis, i.e., we need to investigate means to analyze, characterize and measure software evolvability. In the meantime, we have discovered the insufficiency in the existing software quality models to explicitly address evolvability. For instance, only having a collection of the subcharacteristics of maintainability as defined in the ISO software quality standard [89] 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 and process for analyzing and evaluating software evolvability. This will be described in Chapter 4.

According to Mens and Demeyer [128], 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. A good understanding of the existing software process models is necessary for us to obtain insights in how software changes are integrated in the software development lifecycle.

Knowledge about software architecture, challenges encountered during software evolution, as well as techniques and methods that facilitate software architecture evolution, provides a basic background to architecture evolution. Next chapter will further describe the software architecture evolution research with focus on architecting for software evolvability.

# Chapter 3. Architecting for Software Evolvability

As business and technology evolve and software becomes more complex, software development copes with not only how to create new software systems of the desired quality attributes, but also, following the initial development, how to evolve the systems in their operationally changing contexts. Given that in most cases it is not desirable to develop everything from scratch [128], researchers are constantly challenged to come up with approaches to effectively support the evolution of software systems. For this reason, many research studies have been proposed in this area both by researchers and industry practitioners. These studies focus on how to analyze and improve software evolvability, using a particular technique or practice. However, no systematic review of software architecture evolvability research has been conducted previously to describe the wide spectrum of results in these studies.

The main objective of this chapter is therefore to systematically select and review published literature, and present a holistic overview of the existing studies in analyzing and achieving software evolvability at architectural level.

Secondary objectives are:

- To bring practitioners up to date with respect to the state of research themes that have been actively pursued by the research community in architecting for software evolvability, and quickly identifying relevant studies that suit their own needs;
- To help the research community to identify challenges and research gaps for further exploration.

Concretely, we have stated the following research questions:

- What approaches have been reported regarding the analysis and achievement of software evolvability at the architectural level?

- What are the main research themes covered in the scientific literature regarding analysis and achievement of evolvability-related quality attributes?
- What are the main focus and application contexts of proposed approaches, along with their relevance to software evolvability?
- What is the impact of the studies to research community and practice?

The remainder of this chapter is structured as follows. Chapter 3.1 describes the research method used in this review. Chapter 3.2 presents overview information of the primary studies included in our systematic literature review (SLR). Chapter 3.3 to Chapter 3.7 presents the results of the review in five main categories of themes respectively, with detailed description of relevant studies and analysis of their relevance to software evolvability. Chapter 3.8 discusses the scope of the systematic literature review and validity threats of the review. Chapter 3.9 describes the impacts on research and practice.

## 3.1 Systematic Literature Review Process

This research was undertaken as a systematic review [100] which is a formalized and repeatable process to document relevant knowledge on a specific subject area for assessing and interpreting all available research related to a research question. The research includes several stages:

- Development of a review protocol
- Identification of inclusion and exclusion criteria
- Search process for relevant publications
- Quality assessment
- Data extraction and synthesis

These stages are detailed in the following subsections.

### 3.1.1 Review Protocol

We formulated a review protocol based on the systematic literature review guidelines and procedures proposed by Kitchenham [100]. This protocol specifies the background for the review, research questions, search strategy, study selection criteria, data extraction, and synthesis of the extracted data.

The protocol was mainly developed by me, and was then reviewed by two other senior researchers to reduce bias. The background to the review and the research questions have been described in the beginning of this chapter, while other elements will be explained in the following subsections.

### 3.1.2 Inclusion and Exclusion Criteria

The goal of setting up criteria is to find all relevant studies in our research. We considered full-text papers in English from peer-reviewed journals, conferences and workshops published up to and including the first two quarters of 2010. We did not set a lower boundary on the year of publication because we intended to include all relevant studies that are stored in databases over the years. We excluded studies that do not explicitly relate to software evolution, analysis of software architecture, and software quality that concerns software evolution. We also excluded prefaces, editorials, and summaries of tutorials, panels and poster sessions. Furthermore, when several duplicated articles of a study exist in different versions that appear as books, journal papers, conference and workshop papers, we included only the most complete version of the study, and excluded the others.

A summary of the inclusion and exclusion criteria for this review is presented in Table 3-1. Note that a study must satisfy all inclusion criteria, and not satisfy any of the exclusion criteria.

**Table 3-1: Inclusion and exclusion criteria**

| **Inclusion Criteria** |
|---|
| English peer-reviewed studies that provide answers to the research questions. |
| Studies that focus on software evolution. |
| Studies that focus on software architecture analysis and/or software quality analysis related to software evolvability. |
| Studies are published up to and including the first two quarters of 2010. |
| **Exclusion Criteria** |
| Studies are not in English. |
| Studies that are not related to the research questions. |
| Studies in which claims are non-justified or ad-hoc statements instead of based on evidence. |
| Duplicated studies. |

### 3.1.3 Search Process

We concentrated on searching in scientific databases rather than in specific books or technical reports, as we assume that the major research results in books and reports are also usually described or referenced in scientific papers. However, this does not prevent us from including a book as an identified primary study if the book gives comprehensive descriptions of a certain relevant topic. For instance, the Architecture Tradeoff Analysis Method (ATAM) was described in a conference paper [97], and it was also thoroughly explained in a book [S30][1]. We have therefore included the book as a selected study.

The searched electronic databases include:

- ACM Digital Library (http://portal.acm.org)
- Compendex (http://www.engineeringvillage.com)
- IEEE Xplore (http://www.ieee.org/web/publications/xplore/)
- ScienceDirect – Elsevier (http://www.elsevier.com)
- SpringerLink (http://www.springerlink.com)
- Wiley InterScience (http://www3.interscience.wiley.com)
- ISI Web of Science (http://www.isiknowledge.com).

These databases were chosen as they provide the most important and with highest impact full-text journals and conference proceedings, covering the fields of software quality, software architecture and software engineering in general. After an initial search in these databases, we did an additional reference scanning and analysis in order to find out whether we have missed anything, thus to guarantee that we have selected a representative set of studies. The searched results were also checked against a core set of studies within software architecture evolution and software quality analysis to ensure confidence in the comprehensiveness of search results.

The notion of evolvability is used in many different ways in the context of software engineering with many other closely-related alternative words such as flexibility, maintainability, adaptability and modifiability. Therefore, we consider these words in the list of search terms. In addition, a software evolvability model outlined in [33] identified subcharacteristics that are of

[1] The references starting with S are the studies that were identified in the systematic review. A complete list of these studies can be found in Appendix A.

primary importance for a software system to be evolvable (to be described in Chapter 4). The identified subcharacteristics are a union of quality characteristics that are relevant for characterization of evolution of long-lived software-intensive systems during their lifecycle, comprising analyzability, architectural integrity, changeability, extensibility, portability, testability and domain-specific attributes. Thus, these evolvability subcharacteristics also provided input and motivated the search terms that we used in this research when searching for relevant studies.

Among evolvability subcharacteristics, portability and testability are not explicitly considered as search terms for the review, as we have in the preliminary search found that they are quite often pertained to maintainability, adaptability and flexibility. Domain-specific attribute comprises quality characteristics that are specific for a particular domain, and is considered too general to be used as a search term. The remaining subcharacteristics such as analyzability, changeability and extensibility are included as search terms. In the end, the following search terms were used to find relevant studies, and all these search terms were combined by using the Boolean OR operator:

- S1: software architecture AND evolvability
- S2: software architecture AND maintainability
- S3: software architecture AND extensibility
- S4: software architecture AND adaptability
- S5: software architecture AND flexibility
- S6: software architecture AND changeability
- S7: software architecture AND modifiability
- S8: software architecture AND analyzability

The selection of studies was performed through a multi-step process:

- Search in databases to identify relevant studies by using the search terms;
- Exclude studies based on the exclusion criteria;
- Exclude irrelevant studies based on analysis of their titles and abstracts;
- Obtain primary studies based on full-text read.

Figure 3-1 shows the search process and the number of publications identified at each stage.

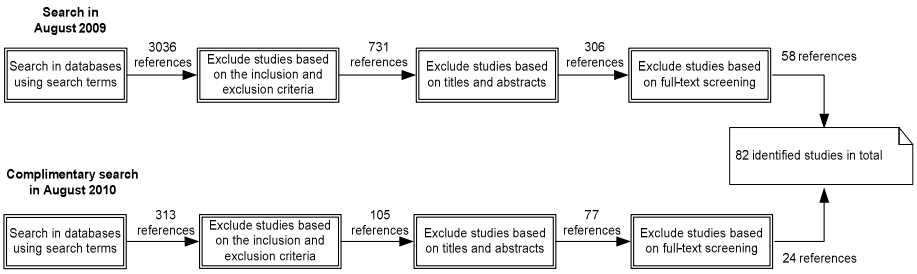


**Figure 3-1: Stages of the search process**

Duplicate publications were removed. We performed the search process at two points in time, i.e., one in August 2009, and the other one in the end of August 2010, with the intention to cover the latest results of publications in 2009 and 2010. In the first search process, the search strategy identified a total of 3036 publications that we entered into the tool EndNote[2], which was also used in the subsequent steps for reference storage and sorting. These publications were checked against the inclusion and exclusion criteria. Irrelevant publications were removed, and this resulted in 731 remaining publications. After further filtering by reading titles and abstracts, 306 publications were left for full text screening to ensure that the contents indeed relate to the topic of software architecture evolution. In the end, 58 studies were identified as primary studies after the first search process. After we had performed a complementary search in the end of August, 2010, following the same entire search process, 24 new papers were added. This resulted in a total of 82 studies in the final list, covering the publications up to and including the first two quarters of 2010. We explain the relative high increase of the studies as: (1) inclusion of studies from 2009 and 2010 (since several studies from 2009 were not available in the database in the first search), and (2) the increased interest in the topic.

### 3.1.4 Quality Assessment

To guide the interpretation of findings in the included studies and determine the strength of inferences, we used the following quality criteria for

[2] www.endnote.com

appraising the selected studies. These criteria indicate the credibility of an individual study when synthesizing results:

- The data analysis of the study is rigorous and based on evidence or theoretical reasoning instead of non-justified or ad hoc statements;
- The study has a description of the context in which the research was carried out;
- The aims of the study are supported by the design and execution of research;
- The study has a description of the research method used for data collection;

To ascertain our confidence in the credibility of a particular identified study and its relevance for data synthesis in the review, all the included studies met each of the four criteria.

### 3.1.5 Data Extraction and Synthesis

The data extraction and synthesis process was carried out by reading each of the 82 papers thoroughly and extracting relevant data, which were managed through bibliographical management tool EndNote and Excel spreadsheets. In order to keep information consistent, the data extraction for the 82 studies was driven by a form shown in Table 3-2.

For data synthesis, we inspected the extracted data for similarities in order to define how results could be encapsulated. The results of the synthesis will be described later in this chapter.

**Table 3-2: Data extraction for each study**

| Extracted Data | Description |
|---|---|
| Identity of study | Unique identity for the study |
| Bibliographic references | Author, year of publication, title and source of publication |
| Type of study | Book, journal paper, conference paper, workshop paper |
| Focus of the study | Main topic area, concepts, motivation and objective of the study |
| Research method used for data collection | Included technique for the design of the study, e.g., case study, survey, experiment, interview to obtain data, observation |
| Data analysis | Qualitative or quantitative analysis of data |
| Application context | Description of the context and application settings of the study, e.g., domain, academic or industrial settings |
| Constraints and limitations | Identified constraints and limitations in the application of an approach as well as the identified areas for future research |
| Architecture-centric activity | Indicating the architecture-centric activity on which the study is focused, e.g., business case, creating architecture, documenting architecture, analyzing architecture, etc. |
| Software lifecycle | The phase of software lifecycle covered in the study |

## 3.2 Scope of the Systematic Review

This systematic review focuses mainly on the studies that describe architectural approaches concerned with software architecture analysis and software quality improvement related to software evolvability. Nevertheless, software evolution spawns also research disciplines that are devoted to the topic of migrating or reengineering legacy software systems by applying a specific software development paradigm or technique to facilitate software evolution, e.g., product line engineering, component-based software engineering, and service-oriented software engineering.

Within the area of software product line engineering, basic principles are elaborated in [28] and [141]. A software product family engineering evaluation model is described by van der Linden et al. [172] to determine the status of software family engineering, concerning dimensions in business, architecture, organization and process. The RE_MODEL method [19] integrates reengineering and product line activities to achieve a transition into product line architecture. The PuLSE method [160] addresses the

different phases of product line development, and is used to systematically analyze a component and improve its reusability as well as maintainability. In order to evaluate the potential of creating a product line from existing products, MAP (Mining Architectures for Product Lines) [166] focuses on the feasibility evaluation process of an organization's decision to move towards a product line. Options Analysis for Reengineering [163] is another method for mining existing components for a product line. Maccari and Riva [118] describe combining reference architecture and configuration architecture to describe legacy product family architecture. Research is also done in domain analysis methods. Some examples of the widely used domain analysis techniques are Feature-Oriented Domain Analysis (FODA) [95] and Feature-Oriented Reuse Method (FORM) [96], which use feature models to organize system features into trees of nodes that represent the commonality and variability within a software product line. Another notation is the orthogonal variability model [14, 141], which is a graph of variation points and variants.

Within the area of component-based and service-oriented software engineering, Jiang and Willey [94] propose a multi-tiered architecture that uses both services and components as architectural elements to offer flexible solutions to the design and integration of large and distributed systems. Wang and Fung [175] propose to organize enterprise functions as services and implement them as component-based systems in order to offer flexible, extensible and value-added services. Cervantes and Hall [49] introduce service-oriented concepts into component models to provide support for late binding and dynamic component availability in the component models. O´Brien et al. [133] explore how service oriented architecture impacts quality attributes. An industrial application of applying these techniques is presented in a white paper [1].

As we see from the above, there are numerous reengineering techniques that help transform software architectures for evolution. However, due to the variety of software development paradigms and the many sub-disciplines concerned in each paradigm, we have chosen to constrain the scope of our systematic review to architectural methods that help analyze and improve software evolvability in general. A survey of the studies that are concerned with the "*what*" perspective [113] of software evolution and various software development paradigms and reengineering techniques that facilitate software evolution remains to be a future work.

## 3.3 Overview of the Primary Studies

A list of all the selected primary studies is provided in Appendix A. This section describes these studies with respect to their sources of publication and citation status which are also indicators on the quality and their impact. A temporal view and research communities that are active in the field of software architecture evolution are presented as well.

### 3.3.1 Data Sources

Most of these 82 studies were published in leading journals, conferences or seminal books that belong to the most cited publication sources in software engineering community. Table 3-3 gives an overview of the distribution of the studies based on their publication channels, along with the number of studies from each source. All the studies fulfill the criteria for quality assessment as described in Chapter 3.1.4. In addition, the impact factor[3] of the publication sources represents also the degree of high quality and potential impact of these studies, and provides confidence in the overall quality assessment of the systematic review. This is also indicated in the citation status described in Chapter 3.3.2.

[3] For instance, based on the search results (performed on 22nd of September, 2010) in respective journal web sites, JSS has impact factor of value 1.34, JST with value of 1.82, Journal of Advanced Engineering Informatics of value 1.73.

**Table 3-3: Study distribution per publication sources**

| Source | Count |
|---|---|
| Journal of Systems and Software (JSS) | 14 |
| Working IEEE/IFIP Conference on Software Architecture (WICSA) | 8 |
| Books | 5 |
| International Conference on Software Engineering (ICSE) | 5 |
| Workshop on Sharing and Reusing Architectural Knowledge-Architecture, Rationale, and Design Intent (SHARK) | 5 |
| IEEE International Conference on Software Maintenance (ICSM) | 4 |
| Journal of Information and Software Technology (IST) | 4 |
| Journal of Systems Engineering | 4 |
| International Conference on Quality Software (QSIC) | 3 |
| International Workshop on Principles of Software Evolution (IWPSE) | 2 |
| IEEE/ACM International Conference on Automated Software Engineering (ASE) | 2 |
| European Conference on Software Maintenance and Reengineering | 2 |
| IEEE International Conference on Engineering of Complex Computer Systems | 2 |
| Journal of Software Maintenance and Evolution | 1 |
| Journal of Systems Architecture | 1 |
| Journal of Computer Standards & Interfaces | 1 |
| Journal of Advanced Engineering Informatics | 1 |
| Journal of Software: Practice and Experience | 1 |
| IEEE International Computer Software and Applications Conference | 1 |
| IEEE International Symposium on Requirements Engineering | 1 |
| IEEE Software | 1 |
| International Conference on Software Engineering Advances | 1 |
| International Conference on Information Science and Applications (ICISA) | 1 |
| International Conference on Research Challenges in Information Science | 1 |
| International Conference on Software Reuse | 1 |
| International Software Metrics Symposium | 1 |
| ACM SIGSOFT software engineering notes | 1 |
| Conference of the Centre for Advanced Studies on Collaborative research | 1 |
| International Conference and Workshops on Engineering of Computer-Based Systems (ECBS) | 1 |
| International Computer Software and Applications Conference | 1 |

| Source | Count |
|---|---|
| ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing | 1 |
| International Workshop on Economic-Driven Software Engineering Research | 1 |
| International Workshop on the Economics of Software and Computation | 1 |
| European software engineering conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering | 1 |
| World Congress on Computer Science and Information Engineering (CSIE) | 1 |
| **Total** | **82** |

### 3.3.2 Citation Status

Table 3-4 provides an overview of the citation rates of the included studies. These numbers are obtained from Google Scholar[4]. The data presented here only gives a rough indication of citation rates, and are not meant for comparison among studies. As shown in the table, 35 studies have been cited by less than 10 other sources. Among these 35 studies, 22 are published in 2009 and 2010, so it is not expected that they can reach a higher citation number in such a short period. Almost half of the studies (38 studies) have been cited by more than 20 other sources. Thirteen studies have very high citation rates with more than 80 other sources.

**Table 3-4: Status of citation rate in detail**

| **Cited by** | < 10 | 10 - 20 | 20 - 30 | 30 - 40 | 40 - 50 | 50 - 60 | 60 - 70 | 70 - 80 | > 80 |
|---|---|---|---|---|---|---|---|---|---|
| **No. of Studies (Total 82)** | 35 | 9 | 10 | 1 | 6 | 4 | 2 | 2 | 13 |

We can see that in general, the citation rates of the studies are quite high, which is also an indicator on the high quality and impact of the studies. We expect that the number of citations will grow since most of the papers have been published in the last six years (see Chapter 3.3.3). The most cited studies (cited by more than 60 other sources) are summarized in Table 3-5.

[4] http://scholar.google.se/ accessed on 4th of September, 2010

The first five studies are books, and the rest are papers in journals and conferences.

**Table 3-5: Most cited studies**

| Ranking | Study | Titles |
|---|---|---|
| 1 | S8 | L, Bass, P. Clements, R. Kazman, Software architecture in practice, Addison-Wesley Professional, 2003. |
| 2 | S27 | L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos, Non-functional requirements in software engineering: Springer, 2000. |
| 3 | S13 | J. Bosch, Design and use of software architectures: adopting and evolving a product-line approach, Addison-Wesley Professional, 2000. |
| 4 | S30 | P. Clements, R. Kazman, M. Klein, Evaluating software architectures: methods and case studies, Addison-Wesley, 2006. |
| 5 | S42 | C. Hofmeister, R. Nord, D. Soni, Applied Software Architecture: A Practical Guide for Software Designers, Addison-Wesley Professional, 2000. |
| 6 | S47 | R. Kazman, L. Bass, G. Abowd, M. Webb, SAAM: a method for analyzing the properties of software architectures, International Conference on Software Engineering, pp. 81-90, 1994. |
| 7 | S48 | R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, The architecture tradeoff analysis method, 4th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 68-78, 1998. |
| 8 | S56 | M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, W. M. Turski, Metrics and laws of software evolution-the nineties view, 4th International Software Metrics Symposium 1997. |
| 9 | S50 | M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, H. Lipson, Attribute-based architecture styles, Working IEEE/IFIP Conference on Software Architecture (WICSA) 1999. |
| 10 | S72 | K. J. Sullivan, W. G. Griswold, Y. Cai, B. Hallen, The structure and value of modularity in software design, 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001. |
| 11 | S11 | P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet, Architecture-level modifiability analysis (ALMA), Journal of Systems and Software, vol. 69, pp. 129-147, 2004. |
| 12 | S46 | R. Kazman., J. Asundi, M. Klein, Quantifying the costs and benefits of architectural decisions, 23rd International Conference on Software Engineering, 2001. |
| 13 | S9 | P. Bengtsson, J. Bosch, Architecture level prediction of software maintenance, 3rd European Conference on Software Maintenance and Reengineering (CSMR), pp. 139-147, 1999. |

| Ranking | Study | Titles |
|---|---|---|
| 14 | S10 | P. Bengtsson, J. Bosch, Scenario-based software architecture reengineering, International Conference on Software Reuse, pp. 308-317, 1998. |
| 15 | S81 | W. M. N. Wan-Kadir, P. Loucopoulos, Relating evolving business rules to software design, Journal of Systems Architecture, vol. 50, pp. 367-382, 2004. |
| 16 | S53 | N. Lassing, P. Bengtsson, H. van Vliet, J. Bosch, Experiences with ALMA: Architecture-Level Modifiability Analysis, Journal of Systems and Software, vol. 61, pp. 47-57, 2002. |
| 17 | S45 | A. Jansen, J. Van der Ven, P. Avgeriou, D. K. Hammer, Tool support for architectural decisions, Working IEEE/IFIP Conference on Software Architecture (WICSA) 2007. |

### 3.3.3 Temporal View

Looking at the studies by year of publication as shown in Figure 3-2, we notice in the trend curve an increasing number of publications in the area of software architecture evolution since 1999. (Note that for year 2010, the review only covers the registered publications in the databases until the first two quarters.) We also notice that all of the included studies were published in 1992 or later. As described in Chapter 3.1.2, we did not set a lower boundary for the year of publication in the search process, yet the time frame of identified studies reflects also the time frame of the evolution and maturation of software architecture area. The significant increase of publications in software architecture evolution area, especially during the last two years, indicates that, as more and more systems become legacy over the years, the crucial role of software architecture evolution is being recognized. The recent boost in research also reflects that the ability to evolve software rapidly and reliably has become a major challenge and research focus for software engineering.

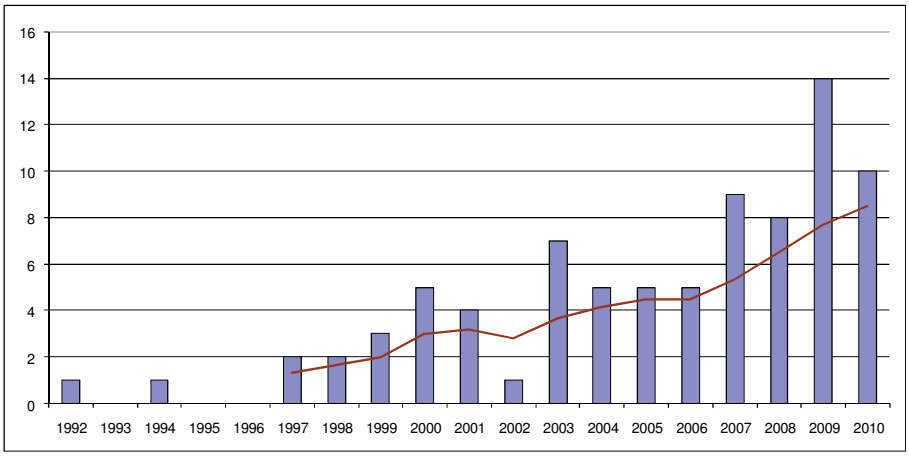


**Figure 3-2: Number of papers by year of publication**

### 3.3.4 Active Research Communities

In terms of the active research communities within the area of software architecture evolution and software evolvability, we look at the affiliation details[5] of the identified set of studies. The assignment of contributed studies of each active research community is based on the affiliations that appeared in the publications. Table 3-6 summarizes the active research communities (with at least two publications within software architecture evolution) along with their corresponding contributed studies. Overall, the set of studies are dominated by Software Engineering Institute (SEI)/Carnegie Mellon University, Vrije University, and University of Groningen.

[5] Please note that during the search process of relevant studies, we did not use any information on authors or research centers for identifying studies because the result of identified studies would be otherwise limited and biased.

**Table 3-6: Active research communities within architecture evolution**

| Affiliations | Contributed Studies | Number of Studies |
|---|---|---|
| Software Engineering Institute, Carnegie Mellon University, USA | [S8, S23, S24, S29, S30, S39, S46, S47, S48, S50, S64] | 11 |
| Vrije University, the Netherlands | [S11, S32, S36, S51, S52, S53, S54, S68, S80] | 9 |
| University of Groningen, the Netherlands | [S13, S32, S43, S44, S45, S53, S78] | 7 |
| University of Texas, USA | [S12, S26, S27, S28, S71] | 5 |
| Blekinge Institute of Technology/ University of Karlskrona/Ronneby, Sweden | [S9, S10, S11, S53, S73] | 5 |
| University Rey Juan Carlos, Spain | [S3, S21, S22, S78] | 4 |
| Swinburne University of Technology, Australia | [S19, S77, S78, S80] | 4 |
| National ICT Australia, Australia | [S1, S77, S82] | 3 |
| University of Limerick, Ireland | [S2, S3, S78] | 3 |
| University of New South Wales, Australia | [S1, S3, S82] | 3 |
| University of Waterloo, Canada | [S47, S58, S74] | 3 |
| Imperial College of Science, England | [S56, S67] | 2 |
| Mälardalen University, Sweden | [S15, S16] | 2 |
| ABB Corporate Research, Sweden | [S15, S16] | 2 |
| Nokia Research Center, Finland | [S33, S34] | 2 |
| Technical University Ilmenau, Germany | [S17, S40] | 2 |
| Texas Christian University, USA | [S18, S35] | 2 |
| University College London, England | [S6, S7] | 2 |

### 3.3.5 Classification of the Primary Studies

As described in Chapter 3.1.5, during the data synthesis phase, we examined the identified studies based on their similarities in terms of research topics and contents in order to categorize the included primary studies of architecture evolution and software evolvability. Besides classifying the included studies, we also examined the research method used for data collection in each study, and application context for each approach described in the studies. The research method used for data collection in the included study is the techniques used for the design of the study, such as case study,

survey, experiment, interview or observation to obtain data. This information is the input to the "Included Technique" columns in Table 3-7 to Table 3-15, explaining the specific techniques used in each approach. The application context of each approach refers to the description of the context and application settings of the study described in the included studies, e.g., domain, academic or industrial settings. This information is the input to the "Validation" columns in Table 3-7 to Table 3-15, explaining the context (academic/industrial setting and in which domain) of the application of each approach.

After examining the research topics addressed in each study, we identified, from the included studies, five main categories of themes, two of which are further refined into sub-categories to group primary studies that share similar characteristics in terms of specific research focus, research concepts and contexts. The categories and sub-categories are:

- **Quality considerations during software architecture design**

  This category focuses on how software quality can be introduced and explicitly considered during software architecture design phase. Three sub-categories are:

  - Quality attribute requirement focused [S8, S10, S13, S25, S26, S27, S79]
  - Quality attribute scenario focused [S24, S30]
  - Influencing factor focused [S1, S29, S31, S38, S42, S80]

- **Architectural quality evaluation**

  This category focuses on the subsequent iteration when the architecture starts to take form, with emphasis on architectural quality evaluation methods that help elicit and refine additional quality attribute requirements and scenarios. Three sub-categories are:

  - Experience based [S14, S34, S37, S50, S73]
  - Scenario based [S11, S33, S47, S48, S53, S54, S62]
  - Metric based [S5, S15, S16, S28, S55, S56, S57, S67, S71, S75]

- **Economic valuation**

  This category focuses on consideration of cost, effort, value and alignment with business goals, when determining an appropriate degree of architectural flexibility. [S4, S6, S7, S9, S18, S23, S35, S46, S64, S66, S72]

- **Architectural knowledge management**

This category focuses on how architecture documentation can be enriched through utilizing different information sources to capture architectural knowledge for quality attributes and their rationale. [S2, S3, S12, S19, S20, S21, S22, S36, S40, S43, S44, S45, S52, S68, S70, S77, S78, S82]

- **Modeling techniques**

This category focuses on modeling traceability and visualizing corresponding impact of the evolution of software architecture artifacts. [S17, S32, S39, S41, S49, S51, S58, S59, S60, S61, S63, S65, S69, S74, S76, S81]

Figure 3-3 illustrates these categories of themes and their corresponding sub-categories along with an overview of distribution of studies.

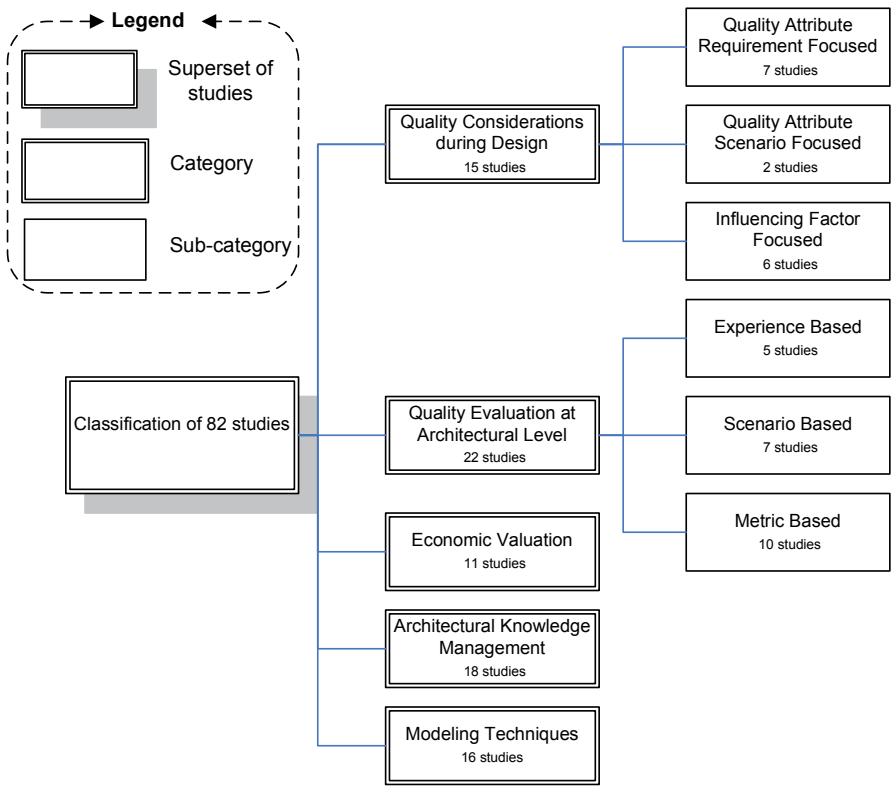


**Figure 3-3: Classification of included studies**

These five categories of themes represent an overview of the main topics of software architecture evolution research. Each theme stands for a research direction on its own, with only a subset of its research and application dedicated to the area of software architecture evolution. As explained, each theme exhibits its specific research focus. Therefore, taking into consideration that evolvability needs to be addressed throughout the complete software lifecycle, the approaches addressed in each category of theme can be combined to complement each other from different perspectives in order to achieve software evolvability.

The categories and their corresponding sub-categories will be further detailed in the rest of this chapter. For each category of theme, we describe the category and related studies, along with their relevance to software evolvability. An analysis of the studies is discussed and summarized in tables. Each table includes the following items:

- The main focus and application context of each approach, including issues such as constraints and limitations;
- The techniques adopted in each approach;
- Research validation environment.

## 3.4 Quality Considerations during Software Architecture Design

This category includes studies that focus on how software quality can be introduced and explicitly considered during software architecture design phase. These studies help identify key quality attributes and constraints early, usually before the software architecture starts to take form. Based on their focus, the studies are further classified into three sub-categories:

- Quality attribute requirement-focused;
- Quality attribute scenario-focused;
- Influencing factor-focused.

### 3.4.1 Quality Attribute Requirement-Focused

The studies in this sub-category perceive quality attribute requirements as the main focus in the software architecture design phase, and consider each design decision based on its implications on the prioritized quality attributes.

- *Attribute-Driven Design (ADD)* [S8] is a recursive top-down method for architects to hierarchically decompose a system and define software architecture by applying architectural tactics and patterns. It is applied after the requirement analysis phase in the lifecycle to accomplish a software system's coarse-grained high-level conceptual architecture. The driving forces in the design process include functional requirements, quality attribute requirements and design constraints that are well-formed and prioritized by stakeholders. ADD centralizes around prioritized requirements. The secondary requirements are fulfilled within the constraints of the most important ones.
- *Quality Attribute Oriented Software Architecture Design (QASAR)* [S10,S13] describes a software design method that explicitly considers quality attributes during the design process. The method consists of three key phases, i.e., functionality-based architecture design, architecture assessment and architecture transformation. The design process starts with an application architectural design based on the functional requirements without explicitly addressing quality requirements. This design is then evaluated with respect to quality requirements qualitatively or quantitatively to achieve an estimated value for each quality attribute. Depending on whether or not the estimated value satisfies the requirement specification, an architecture transformation might be required for quality attribute optimization.
- *Architectural prototyping* [S25] is an another technique to design software architectures by using executable code to investigate architectural quality attributes that are related to stakeholders' concerns with respect to a system under its development.
- *Non-Functional Requirement (NFR) framework* [S27] is a process-oriented and qualitative decomposition approach for eliciting and analyzing non-functional requirements. It systematically takes into consideration the conflicts and synergies between NFRs in order to develop an evolvable architecture. The operation of the framework is visualized through soft-goal interdependency graphs in which quality requirements are treated as soft-goals to be achieved. High-level soft goals are refined into more specific sub-goals. To satisfy each sub-goal, design decisions are reinforced with a design rationale. One limitation of the framework is that it treats all NFRs as soft goals that are to be "satisficed", i.e., not absolutely achieved but within acceptable limits. This might lead to ambiguity in

requirement specifications when there is a need to characterize and quantify hard goals, e.g., requirements in hard real time systems. One example of *using NFR framework along with design patterns* for developing adaptable software architecture is described in [S26]. This approach takes into consideration particular characteristics of software system domain, and refines quality requirements into architectural concepts and alternatives, that are subsequently satisfied with design patterns.

- *Adaptability Evaluation Method (AEM)* [S79] is an integral part of the Quality-driven Architecture Design and quality Analysis (QADA) methodology [122] with specialization in the adaptability aspect. AEM defines adaptability goals through capturing the adaptability requirements that will be subsequently considered in the architecture design. In this study, guidelines on how to model adaptability in architectural models are provided. The approach is used to qualitatively/quantitatively analyze candidate architectures to ensure that adaptability requirements are met before system implementation.

**Relevance to software evolvability**

Except [S26] and [S79], the other approaches address software quality attributes in general, and can be tailored to address evolvability by focusing on evolvability subcharacteristics and by considering the impacts of a design decision on these subcharacteristics. Both [S26] and [S79] explicitly address adaptability, though the definitions of adaptability differ. In [S79], adaptability is regarded as a qualitative property of software architecture's maintainability (which is a superset of flexibility, integrability, testability and modifiability), and includes runtime requirements of the software system as well as adaptation to changes in stakeholders' requirements. In [S26] adaptability is perceived to be heavily dependent on a particular software development project's scope and nature. This approach only focuses on few design patterns that enhance adaptability of real-time software systems, and does not address the multifaceted evolvability perspective of long-lived software systems.

A summary of quality attribute requirement-focused approaches is given in Table 3-7, describing the main focus and application context of each approach, along with issues such as constraints and limitations; the techniques adopted in each approach as well as research validation environment.

Although these approaches focus on quality attribute requirements, they differ from each other. The NFR framework considers quality attributes as soft goals, i.e., there is no clear-cut definition and criteria as to whether they are satisfied or not. This is in contrast with ADD in which quality attribute requirements are well-formed and prioritized. Besides quality attributes, ADD also considers functional requirements as primary drivers in the design process. This is in contrast with QASAR method, which conceives functional requirements as the primary driver for creating application architectural design, whereas quality attributes are treated as secondary drivers, and are not considered a driving force in the first phase of the architecture development.

**Table 3-7. Qualities attribute requirement-focused approaches**

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S8 | Focus on prioritized requirements, i.e., functional requirements, quality attribute requirements and design constraints.<br>Assist architects in making design decisions based on their effects on quality attributes | Recursive top-down design | Validated in various domains |
| S10, S13 | The design process separates architectural design based on functional requirements and quality requirement optimization.<br>An iterative design process to optimize architecture. | Several optimization techniques are used, e.g., scenarios, simulations, mathematical modeling | Validated in embedded systems domain |
| S25 | Investigate architectural qualities and stakeholders' concerns by using executable code. | Experimental technique | Validated in various domains |
| S26 | Require clarifications of the notion of adaptability in order to refine adaptability requirements.<br>Particular domain characteristics are considered. | NFR – soft goal interdependency graph.<br>Design patterns.<br>Qualitative tradeoff analysis of impact | Illustrated by a home appliance control system |
| S27 | Treat non-functional requirements as soft goals.<br>Considers each design decision based on its effects on the quality attributes.<br>Does not provide support to explicitly perform tradeoff analysis between competing design decisions. | NFR framework with soft goal interdependency graph | Validated in various domains |
| S79 | Identify stakeholders and their concerns.<br>Qualitative and/or quantitative analysis of adaptability depending on the knowledge of components' behavior. | Strategic Dependency Model (SDM).<br>Objective reasoning for qualitative analysis | Validated in an industrial case study in wireless environment controlling system |

### 3.4.2 Quality Attribute Scenario-Focused

The studies in this sub-category focus on mapping architectural quality goals into concrete scenarios to characterize stakeholders' concerns throughout the software architecture design phase.

- *Software architecture analysis method* [S24] defines several steps in the software design process: (i) architectural quality goals are expressed through scenarios to characterize the generic quality goals in concern; (ii) mechanisms are tailored to realize the scenarios within the design constraints; and (iii) analytic models are instantiated by scenarios that represent quality attributes of interest or potential risk areas in architecture. The constitution of the analytic models is an iterative process due to the ever-changing architectural requirements and design constraints. As the system evolves, the analytic models can be used to assess the impact of architectural changes and monitor how architectural evolution affects its capability to support predicted modifications.
- *Active Reviews for Intermediate Designs (ARID)* [S30] is a scenario-based assessment method for evaluating intermediate design or parts of an architecture for early feedback. It is a lightweight method that can be used to judge if the design of a partial architecture is appropriate for its intended purpose before the development of the complete architecture. The stakeholders involved in ARID brainstorm and prioritize scenarios that represent foremost problems that the design is expected to handle, in order to assess the suitability of a design approach and discover discrepancies.

**Relevance to software evolvability**

Applying the software architecture analysis approach in [S24] would require quite a number of evolvability scenarios to address and cover evolvability subcharacteristics. Another limitation is that while scenarios are anticipated events in the system life-time, evolvability by nature concerns also unanticipated events. These limitations apply to all scenario-based methods. The approach in [S30] focuses more on scenarios that represent foremost problems the design is expected to handle rather than considering a system's long-term evolvability aspect. Therefore, this approach needs to be complemented with more explicit consideration of scenarios that would cover evolvability concern and subcharacteristics.

A summary of quality attribute scenario-focused approaches is given in Table 3-8. All approaches utilize quality attribute scenarios though with

distinct purposes; the scenarios in [S24] are used for concretizing architectural quality goals, whereas the scenarios in [S30] are used to identify most important functions, issues and problems that are embedded in intermediate design.

**Table 3-8: Quality attributes scenario-focused approaches**

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S24 | Architectural quality goals are mapped into scenarios, mechanisms that realize the scenarios, and analytic models that measure the results | Scenarios. Analytic models | Validated with example scenarios from two real-life software systems |
| S30 | Judge the appropriateness of a partial architecture for its intended purpose during architecture design | Active design review. Scenarios. Stakeholder-centric | Validated in various domains |

### 3.4.3 Influencing Factor-Focused

The studies in this sub-category focus on, early in the design phase, managing factors that are architecturally significant, and constraints that have influence on the design process, along with inter-dependencies among these factors and constraints that would affect the choice of design decisions.

- *ArchDesigner* [S1] is a quantitative quality-driven design approach for architectural design process. The approach evaluates stakeholders' quality preferences and design alternatives. Meanwhile, a software architecture design problem is considered as a global optimization problem due to the inter-dependencies among different design decisions that need to be maintained, as well as global constraints that influence the selection of any design alternative, e.g., project constraints. Optimization techniques are thus used to determine an optimal combination of design alternatives. The influencing factors that are systematically managed are factors that influence the design process, including conflicting stakeholders' quality goals, various design decisions, design alternatives and inter-dependencies, architectural concerns and project constraints.
- *Business goal elicitation* [S29] empowers architects to articulate business goals among stakeholders early in the lifecycle, and is used as prelude to architecture evaluation.

- *Architecture-Based Component composition/Decision-oriented Design (ABC/DD)* [S31] accomplishes architecture design from design decision perspective, by eliciting architecturally significant design issues and exploiting corresponding solutions for these issues.
- *Incorporation of changeability* within a system architecture is a concept introduced in [S38]. It proposes four aspects that have influence on changeability: (i) *flexibility* that characterizes a system's ability to be changed easily; (ii) *agility* that characterizes a system's ability to be changed rapidly; (iii) *robustness* that characterizes a system's ability to be insensitive towards changing environment; and (iv) *adaptability* that characterizes a system's ability to adapt itself to changing environments. These four aspects can be implemented depending on the needed type and extent of changeability.
- *Global analysis* [S42] provides a systematic way to identify and describe architecturally significant factors in the design phase to be able to develop strategies to accommodate these factors, and reflect future concerns early for making design decisions. The influencing factors are classified into three categories: (i) *organizational factors* that constrain design choices; (ii) *technological factors*, such as choices of hardware, software, architecture technology, and standards; (iii) *product factors* that cover a product's functional features and qualities. All these factors interact with each other. They need to be aggregated and prioritized. New factors that may arise during design need to be considered as well. Afterwards, issues that are influenced by these factors are identified, and specific strategies that address the issues are developed to reduce the impact of various factors.
- *Design constraint-oriented approach* [S80] enhances understanding of architectural decision making by treating design constraints, i.e., external forces that restrict an architect's choice of solution space, as central constructs of architecture.

**Relevance to software evolvability**

The ArchDesigner approach in [S1] addresses quality attributes in general, and can be tailored to assess stakeholders' preferences on evolvability subcharacteristics, and determine preferences of design alternatives based on the weighting scores of evolvability subcharacteristics. The Business goal elicitation approach in [S29] is systematic in identifying primary business

drivers for performing an evolvability analysis. Both [S31] and [S80] provide, respectively, a qualitative indication on how the choice of a design decision/design constraints would affect evolvability. The concept in [S38] does not cover the other evolvability subcharacteristics except changeability. The qualities addressed in [S42] emphasize more on operational-related qualities rather than development-oriented quality attributes of a software system such as evolvability. However, identifying organizational factors and technical constraints is relevant to determining strategies in architecting for evolvability.

A summary of influencing factor-focused approaches is given in Table 3-9. All these approaches focus on identifying influencing factors, though with varying perspectives of influencing factors and presence of strengths and weakness. For instance, Global analysis uncovers architecturally significant factors including quality attributes in the early lifecycle of architecture design. There is a clear traceability between influencing factors and derived strategies. But the reasoning about quality consequences of each design decision is not sufficiently supported. This weakness is complemented by [S1], which performs value score computation on stakeholders' preferences on quality attributes and weighting design alternatives' consequences on quality attributes. The Business goal elicitation approach focuses on an organization's business goals, and ties them to quality attribute requirements, whereas ABC/DD [S31] focuses on architecturally significant design issues, and [S80] on design constraints.

**Table 3-9: Influencing factor-focused approaches**

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S1 | Quantitatively determine the optimal design alternative that best satisfy stakeholders' quality goals and project constraints.<br>Observed limitations in judgment uncertainties and judgment consistency. | Interviews. Optimization techniques. Analytic Hierarchy Process (AHP) | Validated as a post-mortem analysis of a production software system for information analysts. |
| S29 | Capture business goals early in the lifecycle. | Business goal scenarios | Validated in Boeing system |
| S31 | Provides an iterative process to implement the architecture design.<br>Issue relationship at different levels is not handled. | Decision abstraction. Issue decomposition principle. | Validated in two large scale projects |
| S38 | Changeability incorporates four aspects, i.e. robustness, flexibility, agility and adaptability. | Theoretical reasoning | Illustrated by examples from varying industries |
| S42 | Identify architecturally significant factors early in the design phase and develop strategies. | Global analysis | Validated in various domains |
| S80 | Identify design constraints and analyze their impact on architecture. | Design constraint properties. | Validated in industrial systems |

## 3.5 Quality Evaluation at Software Architecture Level

An architecture assessment is triggered by various business goals [117], such as evaluating and improving architecture and its qualitative attributes, identifying architectural drift and erosion, identifying risks related to a particular architecture. From an evolution perspective, architecture evaluation is a preventive activity to delay architectural decay and to limit the effect of software aging [170]. The studies in this category focus on quality evaluation at the architecture level when the software architecture starts to take form after the initial design phase. Based on their focus, the studies are further classified into three sub-categories:

- Experience-based evaluation

- Scenario-based evaluation
- Metric-based evaluation.

### 3.5.1 Experience-based

Experience-based architecture evaluation means that evaluations are based on previous experiences and domain knowledge of developers or consultants [12]. The studies in this sub-category focus on extracting experiences of stakeholders and making use of their tacit knowledge. The evaluation process is mostly based on subjective factors such as intuition and experience.

- *Lightweight sanity check for implemented architectures (LiSCIA)* method [S14] focuses on maintainability and reveals potential problems as a software system evolves. This method detects erosion by interviewing system developers using five categories of questions: current grouping of units in modules and future modules, decomposition of functionality over modules, module size, module dependencies, and technologies. The limitations of LiSCIA are: (i) it depends heavily on the evaluator's opinion; (ii) it only aims to discover potential risks related to maintainability; (iii) the use of only a single viewpoint (module viewtype) sets a limit to covering all potential risks.
- *Knowledge-based assessment approach* [S34] evaluates the evolutionary path of software architecture during its lifecycle based on the knowledge of the stakeholders involved in the software development organizations. The extraction of knowledge and factual evidence of claims requires representativeness and completeness in the selection of stakeholders. The drivers for using this method include lack of formal and complete architecture documentation, wide scope of assessment, large number of stakeholders, and geographical distribution of development teams. The outcomes of the assessment are current architecture overview, main issues found, and optionally, recommendations for their resolutions.
- The *concept of identifying causes for changes and strategies to cope with changes* during a system's lifecycle is described in [S37]. This concept is based on analyzing projects that are already finished and extracting experiences on the most frequent changes in terms of sources of stimuli and cost of each change.

- *Attribute-Based Architectural Style (ABAS)* [S50] explicitly associates architectural styles with reasoning frameworks based on quality-attribute-specific models for particular quality attributes. ABAS consists of four parts: (i) problem description that explains the problem being solved by the software structure; (ii) stimuli and response that correspond to the condition affecting the system and measurement of the activity as a result of the stimuli; (iii) architectural styles that are descriptions of component interaction patterns; and (iv) analysis that constitutes a quality-attribute-specific model for reasoning about the behavior of interacting components in the pattern. A specific attribute-based architectural style is accompanied with a set of questions. These questions and answers to the questions are accumulated as a knowledge base that can be exploited during architectural reviews.
- *Decision support method* [S73] quantitatively measures stakeholders' views of the benefits and liabilities of software architecture candidates and relevant quality attributes. The method is used to understand and choose optimal candidate architecture among software architecture alternatives. Although the primary data collection is comprised of subjective judgments, influenced by the knowledge, experiences and opinions of stakeholders, the data collection of stakeholders' subjective opinions is quantifiable. Thus, any disagreements between the participating stakeholders can be highlighted for further discussions.

**Relevance to software evolvability**

The LiSCIA approach [S14] focuses only on maintainability from module viewpoint with respect to dependencies in order to detect erosions, i.e., decreases in architectural structural integrity. Although the knowledge-based assessment approach [S34] addresses evolvability, there is no definition of the authors' perception of evolvability. Lacking explicit consideration of the multifaceted feature of software evolvability, this approach might miss some key aspects that are critical for software evolution. Heavily dependent on stakeholders' subjective interpretation of quality attributes, the decision support method [S73] faces a similar issue. The ABAS reasoning framework [S50] is based on quality-attribute-specific models for particular quality attributes. It does not take into account the tradeoff relationships among quality attributes. Though, in order to determine potential evolutionary paths of an architecture, the preferences and tradeoffs among evolvability subcharacteristics must be considered.

A summary of experience-based quality evaluation approaches is given in Table 3-10. These approaches differ from each other mainly in two aspects:

- Method for data collection

  In [S50, S73], the method for primary data collection is a questionnaire that individual participating domain expert fills out. One possible drawback with a questionnaire is that ambiguous questions might lead to problematic interpretations by participants due to their differing experiences. For instance, [S73] purposely planned to provide less detailed descriptions of architecture candidates in order to provide more room for participants, though with the risk of problematic interpretations of the architecture candidates and relevant quality attributes by participants. As a countermeasure, interviews as in [S14], [S34] and [S37], can be used to complement questionnaires, clarify questions for respondents, capture additional information to the answers from questionnaires, as well as unexpected responses.

- Delivered output of quality evaluation

  The knowledge-based assessment approach in [S34] focuses on identification of key issues that are critical for software evolution. Resolutions to these issues are optional, whereas the decision support method [S73] aims to reach a shared view of resolutions in terms of the choice of architecture candidate by allowing stakeholders to discuss identified disagreements. An accumulated knowledge base for future exploitation is the main output for [S37] and [S50].

**Table 3-10: Experience-based quality evaluation approaches**

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S14 | Detect erosion when it has happened. | Interview.<br>List of questions and actions. | Validated in various industrial domains. |
| S34 | Knowledge-based assessment.<br>Stakeholder-centric: rely on experiences of stakeholders.<br>Implicit iteration in the process.<br>Requires well-focused assessment scope and careful selection of stakeholders. | Semi-structured interviews. | Validated in an industrial mobile terminal product family |
| S37 | Five strategies to cope with change.<br>Prevention and front-loading strategy needs to be complemented with building changeability into system architecture. | Questioning through questionnaire and interviews. | Validated as an exploratory case study in telecommunication domain |
| S50 | Associate a qualitative or quantitative reasoning framework with an architectural style | Questionnaire/checklist. | Validated in various domains |
| S73 | A quantified decision support method that creates increased joint understanding on the choice of software architecture candidates and quality attributes.<br>Risk in problematic interpretation of questionnaire questions, architecture candidates and quality attributes.<br>Rely on experiences of stakeholders.<br>Require sufficient participants to achieve reliable measures. | Questionnaire.<br>Analytic Hierarchy Process (AHP).<br>Discussion meetings. | Validated as an industrial experiment on a software system in automatic guided vehicles system domain with experienced practitioners. |

### 3.5.2 Scenario-based

Scenario-based architecture evaluation means that quality attributes are evaluated by creating scenario profiles for a concrete description of a quality requirement [123]. The studies in this sub-category use scenarios to avoid terminological ambiguities and conflicting interpretation of quality attributes.

- *Software Architecture Analysis Method (SAAM)* [S47, S30] was originally created for evaluating modifiability of software architecture although it has been used for other quality attributes as well, such as portability and extensibility. The primary inputs to the evaluation include system architecture descriptions and scenarios that describe a stakeholder's interaction with the system. Based on these, SAAM establishes a mapping between architecture and the scenarios that represent possible future changes to the system. This mapping provides indications of potential future complexity parts in the software and estimated amount of work related to changes.
- *Architecture Tradeoff Analysis Method (ATAM)* [S48, S30] evolves from SAAM, and evaluates multiple quality attributes for understanding the tradeoffs inherent in the software architecture. It is used to uncover implicit requirements, and reveal how well an architecture satisfies particular quality attributes. It provides insight into how these quality attributes interact with each other by exposing risks, non-risks, sensitivity points and tradeoff points in the software architecture.
- *Holistic Product Line Architecture Assessment (HoPLAA)* method [S62] is an extension to ATAM for assessing product line architecture. This method is performed in two stages to identify risks at two architecture levels: core architecture evaluation, and individual product architecture evaluation. During core architecture evaluation, evolvability points are identified and evolvability guidelines are defined. The notion of evolvability points designates a sensitivity point or a tradeoff point that contains at least one variation point. The identification of evolvability points ensures that quality attributes at individual product architecture level do not conflict with core architecture quality attributes. Evolvability guidelines are used to inform designers about potential conflicts, and guide them to make appropriate design decisions in subsequent product architecture design phase.
- *Architecture Level Modifiability Analysis (ALMA)* [S11, S53, S54] analyzes modifiability based on scenarios that capture future events a system needs to adapt to in its lifecycle. The method consists of five steps: setting analysis goal, software architecture description, change scenarios elicitation, change scenarios evaluation, and interpretation of the results. Depending on the goal of analysis, the output from an ALMA evaluation varies among: (i) maintenance prediction to estimate required effort for system modification to

accommodate future changes; (ii) architecture comparison for optimal candidate architecture; and (iii) risk assessment to expose the boundaries of software architecture by explicitly considering environment and using complex change scenarios that the system shows inability to adapt to.

- A *scenario-based assessment method* [S33] evaluates evolvability of software product line architecture towards forthcoming requirements. The method consists of three phases: (i) scenario collection, classification and prioritization; (ii) architecture evaluation based on the chosen scenarios; and (iii) assessment result compilation. The output includes potential flaws and evolutionary path of the software architecture.

**Relevance to software evolvability**

Both SAAM and ATAM would require quite a number of evolvability scenarios to address all evolvability subcharacteristics. The approaches in [S11], [S53] and [S54] do not cover the other evolvability subcharacteristics except changeability, and thus need to be complemented with other methods to address all evolvability subcharacteristics. In [S33], evolvability of software product line architecture is evaluated towards forthcoming requirements without providing a definition of evolvability. Moreover, this approach provides little guidance in scenario selection, which makes it difficult to develop scenarios that would cover all software evolvability subcharacteristics. The approach in [S62] assesses only product line architecture, and does not focus on the evolution of other types of architecture.

A summary of scenario-based quality evaluation approaches is given in Table 3-11.

**Table 3-11: Scenario-based quality evaluation approaches**

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S11, S53, S54 | Pursue maintenance prediction, risk assessment and software architecture comparison, and focus on modifiability.<br>Goal of the analysis determines techniques for the analysis process, e.g. scenario elicitation technique and scenario evaluation technique. | Interview and brainstormed change scenarios.<br>Scenario classification scheme.<br>Scenario weight estimation. | Validated in various domains |
| S33 | Lightweight analysis method tuned to software product line architecture.<br>Iterative stakeholder-centric process with focus on evolvability.<br>Little guidance to scenario selection and ranking process. | Scenarios.<br>Interviews.<br>Brainstorming session. | Validated in Nokia multimedia software domain |
| S47 | Qualitative assessment.<br>Iterative scenario development.<br>Provide few explicit techniques for the analysis process and relies much on the assessor's experiences. | Brainstormed scenarios.<br>Voting for scenario prioritization.<br>Rank by assigning weights. | Validated in various domains |
| S48 | Identify architectural risks in light of business goals.<br>Consider multiple quality attributes and identify tradeoffs between quality attributes.<br>Explicitly consider both business and technical perspectives.<br>Assess consequences of architectural decisions in light of quality attributes. | Utility tree.<br>Brainstormed scenarios.<br>Voting for scenario prioritization. | Validated in various domains |
| S62 | Focus on risks and quality attributes for both common product line architecture and individual product architecture.<br>Identification of evolvability points and evolvability guidelines.<br>Need further validation and refinement through applying to real life product line architectures. | Utility tree.<br>Brainstormed scenarios.<br>Voting for scenario prioritization. | Demonstrated as an industrial trial |

These approaches exhibit a variety of characteristics. In [S47], the scenarios proposed by stakeholders determine the quality attributes for analysis, whereas in [S48], the quality attributes for analysis are synthesized through explicitly considering both business and technical perspectives. ALMA focuses only on modifiability, and has distinguished analysis goals which determine the choice of change scenarios and techniques used in the analysis process. For instance, for risk assessment, complex scenarios, guided interview and system environment modeling techniques are used; for maintenance cost prediction, scenarios that are likely to occur during the operational lifecycle are used; for architecture comparison purpose, scenarios that are handled differently in architecture alternatives are used. One limitation of the method is that the evaluation of change scenario with respect to its ripple effects on other components relies much on architects' experiences.

### 3.5.3 Metric-based

The studies in this sub-category assess quality impact qualitatively or quantitatively through specific quality metrics.

- Besides *implementation change logs* [S67] and *computation of metrics using the number of modules* in a software system [S56], another set of metrics is based on *software life span and software size* [S75]. Software evolution can also be quantitatively analyzed by using *evolution ratio* which is the amount of evolution in terms of software size, and *evolution speed* which is an indicator of an organization's capability for software system's evolution [S5].
- A *framework of process-oriented metrics for software evolvability* [S71] develops intuitively architectural evolvability metrics, and traces the metrics back to the evolvability requirements based on the NFR framework [54]. Similarly, *process-oriented metric for software architecture adaptability* [S28] analyzes the degree of adaptability through intuitive decomposition of goals and intuitive scoring of goal-satisfying level of software architecture. As the method depends much on intuition and expert expertise, [S57] proposes a *quantitative metric-based approach to evaluate software architecture adaptability*. This approach supports decision-making in choosing architecture candidates that meet stakeholders' adaptability goals that are expressed in scenario profiles. The impact of each scenario profile is measured through IOSA (impact on the

software architecture) and ADSA (adaptability degree of software architecture).

- A *software evolvability model* is outlined in [S15], in which subcharacteristics of software evolvability and corresponding measuring attributes are identified. The subcharacteristics that are of primary importance for long-lived software-intensive systems' evolvability include analyzability, architectural integrity, changeability, extensibility, portability, testability and domain-specific attributes. Measuring attributes for each subcharacteristic are identified as well. The idea with this model is to further refine the identified subcharacteristics to the extent when it is possible to quantify them and/or make appropriate reasoning about the quality of the attributes. Based on this evolvability model, [S16] presents an evolvability analysis method which ensures that the implications of potential improvement strategies and evolution path of a software architecture are analyzed with respect to the evolvability subcharacteristics.
- A *tradeoff analysis method of architecture using architecture analysis and design language* [S55] acquires quantitative values from an architecture model by establishing and measuring metrics of quality attributes.

**Relevance to software evolvability**

Both [S15] and [S16] explicitly address software evolvability, and provides a base and check point for evolvability evaluation and improvement. Both [S28] and [S57] explicitly address software adaptability, i.e., "*the system's ability to make adaptation, which involves environment change detection, system change recognition and system change enactment*" [S28]. The focus of these studies is around changeability subcharacteristic, and does not cover other evolvability subcharacteristics, e.g., analyzability, testability and architectural integrity. Although [S71] focuses on software evolvability, it does not provide any precise definition of evolvability. Instead, the study advocates that the definition and decomposition of evolvability is determined by the domain. This is in conformance to the domain-specific attributes defined in evolvability subcharacteristics.

A summary of metric-based quality evaluation approaches is given in Table 3-12.

**Table 3-12: Metric-based quality evaluation approaches**

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S5 | Base on evolution ratio and evolution speed. | Metrics. | Empirical study in mobile phone software systems |
| S15, S16 | Refine evolvability into seven subcharacteristics that are measured through measuring attributes. | Subcharacteristics and measuring attributes. | Validated in industrial automation domain |
| S28 | Process-oriented qualitative framework for representing and reasoning about adaptability.<br>Depend much on intuition and expert expertise which leads to uncertainty. | NFR framework. | Academic experiment |
| S55 | Quantitatively measure quality attributes for analyzing architecture. | Quality attributes and metrics.<br>Architecture analysis and design language. | Validated in automotive industry |
| S56 | Computation of metrics using the number of modules. | Metrics. | Validated in a financial transaction system |
| S57 | Quantitatively measure and evaluate adaptability through adaptability scenario profile and impact analysis. | Scenario profile.<br>Metrics. | Theoretical reasoning |
| S67 | Base on implementation change logs.<br>More applicable for evaluating maintenance activities instead of evolvability. | Metrics. | Validated with the evolution of kernel of a mainframe operating system |
| S71 | Process-oriented, capture design rationale.<br>Even experienced software engineers need training to do evolvability-related NFR decompositions. | NFR framework. | Two industrial-scale systems, with more than 50000 lines of code |
| S75 | Base on software life span and software size. | Metrics. | Theoretical reasoning |

## 3.6 Economic Valuation in Determining Level of Uncertainty

The uncertainties in software architecture evolution arise from, to a certain extent, understanding how architectural decisions map onto quality attribute responses in terms of costs and benefits. The studies in this category cope with uncertainty in determining an appropriate degree of architectural flexibility and balance with economic valuation to mitigate risks in investment.

- One way to address economic valuation is to estimate the required effort for system modification to accommodate future changes. For instance, *maintenance cost prediction* [S9] calculates the expected effort for each change scenario based on the analysis of how the change could be implemented and the amount of required changed code. The underlying prediction model is based on the estimated change volume and productivity ratios. *Maintenance effort prediction during architecture design* is another method [S4], which takes requirements, domain knowledge and general software engineering knowledge as input to prescribe application architecture, and to quantify maintenance effort by classifying weighted scenarios in terms of complexity.
- Instead of only focusing on cost/effort analysis, *Cost Benefit Analysis Method (CBAM)* [S46] is an architecture-centric economic modeling approach that can address long-term benefits of a change along with its implications in complete product lifecycle. This method quantifies design decisions in terms of cost and benefits analysis, and prioritizes changes to architecture based on perceived difficulty and utility. Another *cost-benefit framework for making architectural decisions* is proposed in [S23]. This approach correlates the change in developer effort to the change in coupling by analyzing a categorized set of modifications to specific software components both before and after an architectural refactoring. *Architecture Improvement Workshop (AIW)*[6] is another method for

[6] There is no publication on this topic yet. Therefore, it is not included in the systematic review. Details on this topic can be found at http://www.sei.cmu.edu/architecture/consulting/aiw/index.cfm (visited on 7th of September, 2010)

taking economic considerations – cost, benefits, and uncertainty, into account by setting values on architectural decisions in relation to quality attributes.

- Software architecture decisions carry economic value in form of *real options* [16, 168]. Options offer flexibility, and allow architectural evolution over time [S6, S35]. A *model for predicting the stability of software architectures using real options* is exploited in [S6], which advocates that the flexibility of an architecture to endure changes in stakeholders' requirements and environment has a value in predicting stability of the software architecture. To maximize the lifetime value of a software architecture, [S35] incorporates the concept of architecture options into design in order to exploit quantitatively an optimal degree of design flexibility. In [S64] the authors hypothesize that architectural patterns carry economic value in the form of real options, and propose to consider cost, value and alignment with business goals to support architectural evolution. This approach guides the selection of design patterns, elicitation of architecturally significant requirements, and valuation of architecture in terms of design decisions with multiple quality-attribute viewpoints. The approach in [S7] provides insights into architectural flexibility and investment decisions related to the evolution of software systems by examining probable changes along with their added value, such as accumulated savings through enduring the change without violating architectural integrity, supporting future growth, and capability of responding to competitive forces and changing market conditions. The approach in [S72] uses *design structure matrices* to model designs and real options technique to value designs.
- Given particular schedule constraints, an *appropriate degree of architectural flexibility* [S66] can be determined through four strategic elements: feature prioritization, schedule range estimation, core capability determination and architecture flexibility determination. The intention is to mitigate the risk of violating schedule, cost and quality constraints.
- Based on several key parameters that have perceived value to a system's stakeholders, [S18] proposes a *conceptual approach to quantify a system's life cycle value to facilitate adaptability* to changes in circumstances and stakeholder preferences.

**Relevance to software evolvability**

Software evolvability concerns both business and technical perspectives as the choice of design decisions when architecting for evolvability needs to be balanced with economic valuation to mitigate risks. Several studies focus on a single quality attribute, e.g., stability in [S6, S7], flexibility in [S35, S66] and modularity in [S72], and do not explicitly consider the multifaceted aspects of evolvability. Both [S46] and [S64] cover multiple quality attributes. However, CBAM relies on the output from ATAM which might not be an appropriate method for extracting scenarios to cover all evolvability subcharacteristics (as explained in Chapter 3.4.2). The approach in [S64] focuses only on the value of architectural patterns for quality attributes that are of interest to stakeholders, and fails to take into account the preferences and tradeoffs among evolvability subcharacteristics. A summary of economic valuation approaches is given in Table 3-13.

**Table 3-13: Economic valuation approaches**

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S4 | Predict maintenance efforts at architectural level. | Growth scenario profile.<br>Scenario classification with respect to complexity. | Validated with a web content extraction application architecture |
| S6 | Value flexibility and view stability as a strategic architectural quality that adds values in form of growth options. | Real options theory | Theoretical reasoning |
| S7 | Provide insight into architectural stability and software evolution investment decisions. | Real options theory | Validated in an academic experiment of a refactoring case study |
| S9 | Augment architecture description with size estimates.<br>Prediction of maintenance efforts.<br>Dependency on domain experts and architects.<br>Lack of validation the representativeness of a maintenance profile. | Change scenarios.<br>Prediction model. | Exemplified in the medical equipment domain |

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S18 | Quantify lifecycle value of enduring systems. | Surveys and interaction with stakeholders.<br>Market surveys and user group assessment. | Exemplified with a cellular telephone system |
| S23 | Correlate change in developer effort to the change in coupling.<br>Compute predicted savings in effort. | Compute average change in coupling and effort | Validated in a marketing services company |
| S35 | Static and dynamic evaluation of architecture flexibility. | Real options theory.<br>Metrics.<br>Optimization techniques. | Illustrated with quantitative examples |
| S46 | Analyze cost and benefits of architectural strategies.<br>Sensitivity to uncertainty in cost and benefit values.<br>Rely on ATAM to identify architecture strategies. | Quality attributes scores.<br>Benefit and cost quantification. | Validated in various domains |
| S64 | Consider cost, value and alignment with business goals when exploiting option values of an architectural pattern. | Real options theory | Theoretical reasoning |
| S66 | Model-based approach to assist in determining an appropriate degree of architectural flexibility within constraints.<br>Need further calibration and validation of architecture flexibility determination model. | Expert judgment.<br>Parametric cost modeling. | Academic experiment in a full text system |
| S72 | Modularity in design creates value in the form of real options.<br>Model design and value the design. | Data structure matrices.<br>Real options theory. | Illustrated with Parnas' KWIC example |

All these approaches consider at least one of the following, i.e., cost, effort, value and alignment with business goals, when determining an appropriate degree of architectural flexibility.

## 3.7 Architectural Knowledge Management

The studies in this category focus on utilizing various information sources to capture architectural knowledge, which is comprised of architecture design, design decisions, assumptions, context, and other factors that together shape a software architecture. In spite of the exhibited properties of large software systems as described in Brooks' study [38], e.g., software complexity, inevitable changes of software systems and invisibility of software structure representation, architectural integrity needs to be maintained. An explicit representation of architectural knowledge is therefore necessary for evolving systems and assessing future evolutionary capabilities of a system [106].

- Apart from using change scenarios and change cases to model variability and describe future evolutionary capabilities, it is also useful to *explicitly model invariability assumptions*, i.e., things that are assumed will not change [S52]. Assumptions are design decisions and rationale that are made out of personal experience and background, domain knowledge, budget constraints and available expertise. There are three types of assumptions: *technical assumptions* that concern the technical environment a system is running in, *organizational assumptions* that concern the organizational aspects in a company, and *managerial assumptions* that reflect the decisions taken to achieve business objectives. The discovery and recovery of architectural knowledge in terms of assumptions help assess the evolutionary capabilities of system architecture. These assumptions can also be used to provide additional what-if scenarios for software architecture assessment, i.e., what if a certain assumption proves to be invalid. In addition, explicit representation of traceability between architecture evolution and early-made assumptions would supplement design decisions to confront uncertainties when predicting future user requirement changes. A relevant method is *Recovering Architectural Assumptions Method (RAAM)* [S68] that makes assumptions explicit by recapitulating historical information of software system evolution.

- To assess architectural design erosion [174], an *architecture assessment model* measures the extent of deviation in terms of functional and structural divergence [S12]. In order to track software evolution, the loss of system functionality and architectural structure are represented using functional and structural erosion indicators respectively, indicating whether changes that are incorporated into a system would violate integrity of architectural design.
- As architectural constraints influence the quality of architectural design process and improvement of software quality, the *concept of classifying architectural constraints* [S40] is used to generalize architectural styles and patterns.
- *Documenting architectural design decisions (ADD)* is another approach to maintain architectural artifacts in order to evolve software in a controlled way without compromising software integrity [21]. [S77] reports on practitioners' perception of the value, usage and documentation of design rationale, and argues for the need of tool support for capturing and using design rationale to avoid knowledge vaporization and dependency on domain experts. In line with this reclamation, several tools have been developed [S2, S3, S20, S21, S22, S36, S43, S44, and S45] for sharing design decisions along with rationale. [S19, S70, S78] provide comparative studies of these architecture knowledge management tools. [S70] suggests another tool for visualization of design decisions and rationale, in order to overcome the deficiencies in the existing tools, e.g., visualization support for dependency relationship between ADDs, support for collaborate decision-making, and rationale visualization support.
- *Mining patterns* to systematically extract and document architecturally significant information [S82] improves architecture evaluation activities for pattern-oriented systems. General scenarios and architectural tactics are extracted from software patterns, and are used as input to architecture evaluation, and vice versa, the architecture evaluation results provide input to pattern validation.

**Relevance to software evolvability**

The studies in this sub-category focus on capturing architectural knowledge, and therefore are useful in improving architectural integrity which is one of the evolvability subcharacteristics.

A summary of architectural knowledge management approaches is given in Table 3-14.

**Table 3-14: Architectural knowledge management approaches**

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S2, S3 | Capture design decisions and rationale for quality attributes, and provide knowledge repository.<br>Explicitly augment quality attribute utility tree with design decisions.<br>No support on diagrammatic modeling of design decisions.<br>Need to be integrated with requirement management tool to avoid work duplication. | Open source groupware platform, i.e., Hipergate.<br>Data model. | Validated as an industrial trial in architecture evaluation process |
| S12 | Objectively measure the extent of architectural deviation in the system.<br>Might have limitations in handling large scale legacy system. | Abstract architectural model representation.<br>Architectural erosion measures. | Validated in a sample university registration system |
| S20 | Capture design decisions and rationale for functional requirements.<br>Less attention is paid for recording quality attribute knowledge. | Argumentation representation.<br>Argument ontology. | Validated in a set of experiments |
| S21, S22 | Provide support for capturing design decisions for quality attributes and their rationale.<br>Describe and document explicitly tacit knowledge.<br>Selection of mandatory and optional attributes for capturing design decisions. | Mandatory and optional attributes. | Validated in a virtual reality system |
| S36 | Integrated functionality of architectural knowledge sharing supports architects in decision-making process.<br>Less attention is paid for recording quality attribute knowledge. | Architectural knowledge sharing portal. | Validated as an experiment in a software development organization |
| S40 | Capture high level architectural design knowledge.<br>Cover only a subset usage of architectural styles. | Taxonomy based on ANSI/IEEE 1471 standard. | Theoretical meta study based on empirical research results |

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S43 | Add formal architectural knowledge (AK) through annotating the existing documented AK sources based on a formal meta model. | Domain model.<br>Formal meta-model.<br>Plug-ins. | Validated through a large industrial example |
| S44 | Iterative process of recovering architectural design decisions.<br>High dependency on architects for the recovery process. | Tool support. | Validated in an academic experiment |
| S45 | Tool support at the later stages within design to bind architectural decisions, models and system implementation.<br>Less attention is paid for recording quality attribute knowledge.<br>Not explicitly address design decision evolution perspective. | Architectural description language integrated with Java. | Validated in an academic experiment |
| S52, S68 | Utilize different information sources to capture assumptions in order to assess the architecture's evolutionary capabilities.<br>Evolutionary aspects of assumptions are not addressed. | Source code access.<br>Historical information.<br>Interviews.<br>Documentation. | Validated in an e-commerce software product |
| S70 | Support explicit rationale visualization of an architectural design decision. | Argumentation-based approach. | Not validated yet |
| S77 | Empirical investigation of use and documentation of design rationale. | Surveys. | A survey of practitioners |
| S19, S78 | Comparative study of architectural knowledge tool support. | Comparison framework of 10 criteria. | Not applicable |
| S82 | Improve software architecture design and evaluation through mining patterns.<br>Initial work on improving architecture evaluation activities for pattern oriented systems. | Scenarios.<br>Tactics. | Validated in an academic demonstration by using EJB architecture usage patterns |

To achieve a good understanding of decisions that sustain an architecture, [S52, S68] capture assumptions that architectural decisions are often based on. [S20, S36, S44, S45] focus specifically on capturing and managing

design decisions and rationale for functional requirements, whereas [S2, S3, S21, S22] pay more attention to capturing quality attributes knowledge, i.e., design decisions and rationale for quality attributes. [S20, S22] further distinguishes from other studies with its explicit emphasis on architecture views. [S44, S45] consider software architecture as a composition of a set of architectural design decisions. [S44] focuses on recovering architectural design decision for the purpose of reverse engineering, whereas [S45] maintains the relationships between design decisions for the purpose of forward engineering. Both approaches have a similar architectural design decision model, though [S45] extends the decision model by combining it with a meta-model that is comprised of an architectural model, a requirement model and a composition model. This allows architects to document architectural design decisions with traceability to related requirements and part of the implementations. However, the evolution perspective is not explicitly addressed in [S45]. Besides codifying architectural knowledge that concerns an architecture, [S36] distinguishes from the above mentioned studies with one supplementary feature, i.e., architectural knowledge sharing using personalization techniques.

## 3.8 Modeling Techniques

Due to the fact that all artefacts produced and used during the entire software lifecycle are subject to change, the studies in this category mainly focus on modeling artifacts to support software architecture evolution.

- *Modeling traceability links* between requirements, features, architectural elements and implementation is described in [S17] to improve evolvability. A formal definition of indicators that concern evolvability deficiency and corresponding resolution actions is provided as well.
- To assess software architectures for evolution and reuse, a *framework in modeling relevant information and architectural views* [S58] is proposed for reengineering, analyzing, and comparing software architectures. The types of information for traceability modeling include: (i) *stakeholder information* that describes stakeholders' objectives, and provide boundaries for analysis; (ii) *architecture information* such as design principles or architectural objectives; (iii) *quality information*; and (iv) *scenarios* that 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 assess the architecture's support for potential enhancements. In this way, sensitivity points of a system are revealed. A *lightweight traceability management concept* [S51] proposes to customize traceability by scoping the traces that need to be maintained to only activities stakeholders must carry out.

- The approach in [S63] focuses on *managing quality properties during the whole lifecycle of model-driven development*. Besides using model and quality-driven architecture design/evaluation, this approach is extended with knowledge engineering, and involves three main phases: modeling reusable quality requirements, representing quality in architectural models, and model-based quality evaluation on whether the desired quality goals are met in models and code.
- *Using architectural tactics to embody non-functional requirements (NFRs) into software architecture* is described in [S49]. These tactics are reusable architectural building blocks that provide generic solutions to quality attribute issues. The tactics along with their relationships are represented in *Feature models*, whereas the structure and behavior of tactics are described using the *Role-Based Modeling Language (RBML)* [99]. Another tactic-based modeling is *tactic-based non-functional requirement (NFR) modeling approach* [S59], which incorporates NFRs into software analysis and design phase. Based on a classification framework of tactics types, the approach focuses on tactics of NFRs rather than the NFRs themselves, and manages tradeoffs among competing NFRs by considering prioritization and impact of tactics on NFRs.
- A *concern-driven software development approach* [S61] supports developers in understanding and evolving software systems. A concern is a concept that relates a group of software fragments. The approach consists of three main elements: (i) a fine-grained concern model that associates each concern to the set of artifacts that implement the concern; (ii) visualization of concerns at both code level and architectural level; and (iii) automated support in maintaining concern model over time.
- Formalizing and modeling architectural knowledge is essential for understanding the resulting impact on architectures and software systems. One way to model architectural knowledge is based on *ontology*, as ontology can be used to formally define and capture architectural knowledge, e.g., architectural design decisions, and

architectural styles. Thus, ontology mechanisms provide a conceptual modeling and reasoning support for architectural knowledge modeling, which helps to determine essential aspects in managing architecture evolution. The approach in [S32] *uses ontology to visualize architectural design decisions* by means of scenarios such as quality attribute tradeoff analysis, impact analysis and if-then scenarios. Another *ontological approach for architectural style modeling* [S65] is based on description logic. Instead of using ontology to model architectural style, [S76] proposes to evolve software architecture by using *graph transformations* to provide a formal specification of evolution patterns.

- *Modeling an evolvable system by building a wrapper-system* [S60] coordinates three stages of iteration: capturing system behavior, updating system state, and applying new changes. By using a *clustering algorithm*, [S69] identifies software layers for understanding and evolution of object-oriented software systems. To allow architects to precisely express and reason about architecture evolution with the goal of choosing an optimal evolution path for an architecture, [S39] focuses on (i) *evolution path*, which is a first-class entity for representation and analysis; and (ii) *evolution style*, which defines a family of domain-specific architecture evolution paths that share common properties and satisfy a common set of constraints.
- *Modeling change impact* [S41] between software architecture and its related source code is performed by using (i) Architectural Software Component Model (ASCM) which represents software architecture descriptions; (ii) typology of change operations; (iii) formalized change propagation mechanism; and (iv) defined change propagation process.
- To address evolution of system requirements and software architecture, *quality-driven software reengineering model* [S74] adopts NFR Framework [54] and the concept of soft goals to support modeling of design rationale with soft-goal interdependency graphs.
- The approach in [S81] focuses on *business rules*, which represent an important source of requirement changes due to their high impact on software and business process. Business rules are considered as an integral part of system evolution, and are specified in Business Rule Model, which is then related to meta-model level of software design

elements through a Link Model. Modeling business rules improves requirement traceability in software design, and helps in localizing impacts of changing business rules.

**Relevance to software evolvability**

The modeling-techniques help improve architecture evolution by modeling the relationships among inter-dependent software artefacts, which if not handled with care, would introduce inconsistencies and lead to evolvability degradation in the long run.

A summary of modeling techniques is given in Table 3-15.

**Table 3-15: Modeling techniques**

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S17 | Model relations between requirements, features, architectural elements and implementation for evaluating and improving evolvability. | Traceability modeling.<br>Features. | Validated in an industrial IT infrastructure domain |
| S32 | Model architectural design decisions using ontology-driven visualization. | Ontology instances. | Validated in a product audit organization |
| S39 | Model evolution paths with the goal of choosing an optimal path to achieve business objectives.<br>Characterize recurring patterns as a set of evolution styles. | Utility-theoretic approach. | Theoretical. |
| S41 | Model change impact on the structure of software architecture. | Rule-based approach. | Implementation based on Eclipse Development Environment. |
| S49 | Model architectural tactics in feature models, and define semantics for these tactics. | Feature modeling.<br>Role-based meta-modeling language | Demonstrated with a stock trading system. |
| S51 | Scope for a minimum set of links to model traceability. | Traceability path | Illustrated with examples in product line engineering and process management |
| S58 | Model information of stakeholder, architecture, quality and scenarios.<br>Risk level indication through estimating the required effort (low, medium, or high) to make the changes.<br>Analysis is based on stakeholder objectives, and requires upfront modeling and compilation of various stakeholders' perspectives. | Traceability modeling.<br>Scenarios.<br>Architecture views.<br>Quality function deployment. | Empirical study in a large scale telecommunication switching system |
| S59 | Model tactics as opposed to focusing on NFRs themselves. | NFR framework.<br>Qualitative and quantitative analysis. | Illustrated with a case study of Automatic Teller Machine (ATM) application |

| Study | Focus and Application Context | Included Technique | Validation |
|---|---|---|---|
| S60 | Construct a wrapper system which generates feedback data, and detects the need for evolutionary changes. | Object-process modeling. | Validated by analyzing system usage activity logs and update request history of projects |
| S61 | Model concerns and map them towards software artifacts. | Concern model. | Three small evaluations assessing different aspects |
| S63 | Model quality requirements to create quality attribute ontology and requirements models.<br>Quality driven model selection from architectural knowledge base.<br>Model based quality evaluation (qualitative and quantitative) | Ontology. Model-driven engineering. Domain specific modeling.<br>Scenarios.<br>Quantitative measuring techniques, prediction methods, measurement based methods. | Validated in a secure middleware project |
| S65 | Conceptual modeling of architectural styles. | Ontology.<br>Description logic. | Illustrated with an example. |
| S69 | Identify software layers for the understanding and evolution of existing object-oriented software systems. | Clustering algorithm. | Empirical investigation |
| S74 | Model NFR requirements to guide software transformation.<br>Not explicitly address the estimation of transformation impact. | NFR framework.<br>Soft-goal inter-dependency graphs.<br>Design patterns. | Validated with two medium-size software systems (less than 9 KLOC) |
| S76 | Use sequences of architectural restructurings to specify architecture evolution. | Graph transformations. | Validated with an Internet shop application |
| S81 | Model business rules as an integral part of a software system evolution.<br>Improved traceability between requirements and design. | Model.<br>Typology. | Validated in a healthcare information system |

## 3.9 Impacts on Research and Practice

The identified categories of themes provide an overview of software architecture evolvability research as well as a basis for discovering possibilities for improvement in research and practice. This section summarizes a number of implications for research and practice.

### 3.9.1 Technology Maturation

This systematic literature review provides us a perspective of where the field of architecture evolution and software evolvability stands today. To get better understanding of the development of the field, we examined the maturity phase of the approaches described in the primary studies by mapping them against Redwine-Riddle model [147], which identifies six typical phases for technology maturation, typically taking 15-20 years for a technology to enter widespread use.

- *Basic research*

This is a phase of investigation of ideas and concepts, and articulation of research questions;

- *Concept formulation*

This is a phase of informal circulation of ideas and convergence on a compatible set of ideas;

- *Development and extension*

This is a phase of exploration of preliminary use of the technology, clarification of underlying ideas, and generalization of the approach;

- *Internal enhancement and exploration*

This is a phase of extension of the general approach to other domains, usage of the technology to solve real problems, and stabilization of the technology;

- *External enhancement and exploration*

This is a phase of involvement of a broader group outside the development group to show substantial evidence of value and applicability of the technology;

- *Popularization*

This is a phase of appearance of production-quality, supported versions and commercialization of the technology.

I and two other senior researchers (the same researchers participating in the whole systematic literature review process) reviewed the 82 primary studies, and cataloged independently the maturation classification of the technology presented in each study. When there were any discrepancies in the judgment on maturation level of any studies, discussions were then initiated in order to reach an agreement. Figure 3-4 summarizes the classification results[7] (number of studies indicated in parenthesis for each maturation phase and maturation distribution in percentage) according to the technology maturation model.

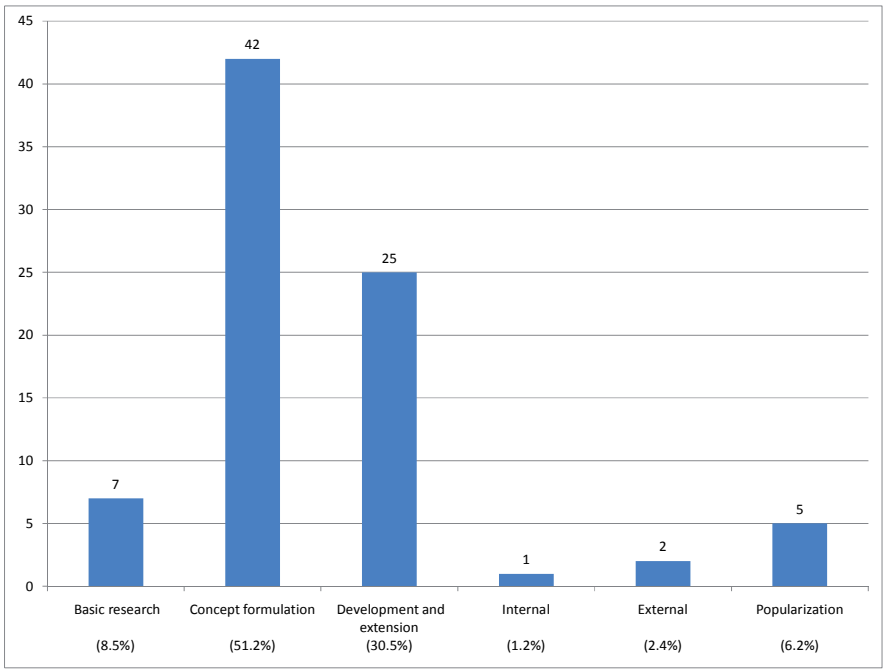


**Figure 3-4: Technology maturation classification of primary studies**

We can see from the classification result that a large majority of the 82 primary studies belong to early maturity stages; almost 60% of studies belong to early stages (basic research and concept formulation), while

[7] Figure 3-4 is based on the data collected from peer-reviewed journals, conferences and workshops, which are the sources in focus in our research. Considering that some of the later elements of the model would be perhaps found in white papers, industry conferences, and company technical reports, there might be some variation if we expand the scope of data sources.

around 30% of studies come to the development and extension phase. This implies that most methods and tools are still not widely established in industrial practices, indicating that the value and applicability of many novel research ideas still need to be further extended on industrial projects of various scales and in different industrial domains.

### 3.9.2 Theoretical Foundation and Formalization

The 82 primary studies concern two main aspects:

- Development of new, or modification of existing approaches to support architecture evolution and software evolvability;
- Evaluation of the effect of applying an approach.

To get a good understanding of how the approaches have been assessed, we examine the primary studies by looking into the empirical method they use, e.g., theoretical reasoning, single-case validation in industry, etc.

A distribution of the studies per validation status is shown in Table 3-16.

**Table 3-16: Study distribution per empirical method used**

| Empirical Method | Categ. 1 | Categ. 2 | Categ. 3 | Categ. 4 | Categ. 5 | Number | % |
|---|---|---|---|---|---|---|---|
| Single-case in Industry | 3 | 12 | 5 | 6 | 8 | 34 | 41.5 |
| Single-case in Academia | 1 | 1 | 2 | 6 | 5 | 15 | 18.3 |
| Multiple-case | 9 | 7 | 1 | 0 | 1 | 18 | 21.9 |
| Theoretical Reasoning | 1 | 0 | 3 | 4 | 2 | 10 | 12.2 |
| Survey | 1 | 2 | 0 | 2 | 0 | 5 | 6.1 |
| **Total** | 15 | 22 | 11 | 18 | 16 | **82** | **100%** |

**Note**:

**Categ. 1** Quality consideration during design. **Categ. 2** Quality evaluation at architectural level. **Categ. 3** Economic valuation. **Categ. 4** Architectural knowledge management. **Categ. 5** Modeling techniques.

About one-fifth (21.9%) of the studies have extended their approaches for solving industrial problems in multiple domains. Two out of the five surveys were conducted on practitioners in companies. Most of the case studies are single-case, with 34 studies done in projects in industry and 15 studies in academic settings. Moreover, 8 studies are on theoretical level, indicating

also the challenge in collecting empirical data due to the complex and longitudinal nature of software evolution. As we see from the table, 63.4% (i.e., 41.5% + 21.9%) of the studies include industrial case studies, and 71.7% (i.e., 41.5% + 18.3% + 21.9%) include case studies. This large percentage of case studies implies:

- Software evolution research studies real-world phenomena, and the knowledge is acquired on the basis of case studies rather than deductive logic, mathematics, or generalized knowledge, as generalizing the results from case studies to settings beyond the studied organizations is a challenge;
- Architecture evolution and software evolvability is less expressive in formalized ways (foundation theories, quantitative methods, formal languages);
- Software evolution research area, by its nature, due to its complexity, is more difficult to be explained by theoretical principles than by practical experiences; thus, a theoretical foundation with practical value for software evolution is necessary.

### 3.9.3 Combination of Approaches

Each of the approaches identified in the review has its specific focus and context that it is appropriate for. For instance, the Attribute Driven Design (ADD) [S8] assists in making design decisions based on their effects on quality attributes. The input to its commencement depends on some analysis results from other methods, e.g., Quality Attribute Workshop (QAW) [8] which helps in understanding the problem by eliciting quality attribute requirements in the form of quality attribute scenarios. Moreover, ADD uses prioritization of quality attributes when the choice of architectural patterns and tactics cannot support all the desired quality attributes. In this context, ADD depends on some kind of architecture evaluation method, e.g., ATAM [S30, S48], in order to analyze how each design alternative would influence the tradeoffs among all desired quality attributes. Therefore, considering the architectural design activities in the software lifecycle, ADD needs to be

[8] There is no publication on this topic in the electronic databases. Details on this topic can be found at http://www.sei.cmu.edu/architecture/consulting/qaw/index.cfm (visited on 22nd of September, 2010)

complemented with approaches that support elicitation of quality requirements as well as approaches that support reasoning about choice of design alternatives.

Another example is related to scenario-based analysis methods. Most scenario-based software architecture analysis methods have the strength of being able to concretize driving quality attribute requirements, but they also have a weakness of being optimistic in change scenario elicitation due to the unpredictable nature of changes as well as stakeholders' short horizon in foreseeing future changes [110]. Therefore, some architectural knowledge management approaches can be used to complement scenario-based methods and address this weakness through explicit representation of invariabilities to provide additional what-if scenarios. Economic valuation methods can also be used to complement with details on business consequences of architectural decisions. Another weakness of most scenario-based analysis methods is their lack of a more fine-grained analysis [S58] although most of these approaches are effective for high-level evaluation of an architecture. Modeling techniques can thus be used to complement with traceability information and visualization of impact analysis.

We have observed an initiative in research community to combine appropriate techniques for software architecture evolution [64, 131]. As evolvability needs to be addressed over the complete software lifecycle, it is necessary to combine appropriate approaches to manage this multifaceted attribute [S15].

### 3.9.4 Tailoring Approaches for Specific Contexts

For practitioners, this review presents a wide spectrum of approaches that analyze and improve software evolvability from specific perspectives. As described in Chapter 3.9.3, each approach identified in the review has its specific application context that it is appropriate for, such as the required input for commencement when using an approach, the phase in the software lifecycle when an approach is suitable, scope of analysis and output, etc. Thus, this review can be used by practitioners as a source in searching for relevant approaches. We suggest that the main consideration for practitioners is to carefully examine the context and characteristics of their own project, and compare with the application context and constraints of a certain approach before adopting and tailoring the approach into their own software development.

## 3.10 Summary

The main objective of this chapter is to obtain a holistic view of and critically analyze the existing studies in analyzing and achieving software evolvability at architectural level. Based on a pre-defined search strategy and a multi-step selection process, we have identified 82 primary studies, covering a spectrum of approaches with specific perspective or focus on a particular architecture-centric activity in software lifecycle. These approaches vary in terminology, descriptions, artifacts and involved activities, yet beyond these differences, we find approaches that share a lot in common, e.g., focus, goal and application context. We extract these commonalities and summarize the studies into five main categories of themes:

- **Quality considerations during design**

The approaches in this category are further refined into three sub-categories:

  - Quality attribute requirement focused
  - Quality attribute scenario focused
  - Influencing factor focused

Most of the techniques that support quality considerations during software architecture design help identify key quality attribute requirements early in the software design phase. Most studies address quality attributes in general and not evolvability in particular.

- **Architectural quality evaluation**

In the subsequent iteration when an architecture starts to take form, architectural quality evaluations help elicit and refine additional quality attribute requirements and scenarios. The approaches in this category are further refined into three sub-categories:

  - Experienced-based
  - Scenario-based
  - Metric-based

A reflection on how these studies are related to software evolvability is that most studies focus on particular quality attributes such as adaptability, and do not cover the wide spectrum of evolvability subcharacteristics. Few studies explicitly address software evolvability. Even if the term evolvability is used in some studies, there is a lack of

precise definition or explanation of authors' perception on software evolvability.

- **Economic valuation**

Economic valuation approaches provide more details on architectural decisions' business consequences, and assist development teams in choosing among architectural options. Most studies focus on a single quality attribute, e.g., stability, flexibility or modularity, and may exhibit a drawback in architectural design decision-making process when multiple evolvability subcharacteristics are involved, requiring explicit management of preferences and tradeoffs among evolvability subcharacteristics.

- **Architectural knowledge management**

Architectural knowledge management approaches improve architectural integrity by enriching architecture documentation with architectural knowledge captured from different information sources.

- **Modeling techniques**

Modeling techniques add value by modeling software artefacts along with their traceability, and visualizing corresponding impact of the evolution of software architecture artifacts. They do not explicitly focus on evolvability in particular, but they help control and improve software architecture evolution by modeling the relationships among inter-dependent software artefacts.

This systematic review has implications for both research and practitioners. For researchers, the analysis of the primary studies indicates a number of challenges and topics for future research:

- There is a space to develop new foundation theories beyond to Lehman's law (for example quantitative expression of evolvability, along with its measurement, monitoring, prediction, impact analysis, and similar), with practical value to software architecture evolution; In future we can expect more research work in this area – in addition to case studies, we could expect more basic foundation research and standardization of designing, and assessing evolvability, probably enriched by different tools.
- It is necessary to address the multifaceted perspectives of software evolvability through combining appropriate approaches to complement each other, as each approach has its specific focus and context that it is appropriate for in a software lifecycle;

- Considering that all artefacts produced and used during the entire software lifecycle are subject to changes, novel methods and tools need to be developed to be able to design ultra-large-systems that integrate and orchestrate the evolution of thousands of platforms, decision nodes, organizations and processes [132].

For practitioners, they can use this review as a source in searching for relevant approaches before adopting and tailoring them by examining the context and characteristics of their own software development, and comparing with the application context of relevant approaches.

The analysis of the existing studies in analyzing and achieving software evolvability at architectural level lays a ground for our research in evolvability analysis; in particular, the multifaceted characteristics of evolvability as well as the different theme to address evolvability provide us with valuable input to our evolvability analysis process, which will be described in the next chapter.

# Chapter 4. Analyzing Software Evolvability

Architecting an evolvable software system is an important and challenging task. This is mainly due to the following reasons:

- Change is an essential element in software development, as software systems must respond to evolving requirements, platforms and other environmental pressures [80].
- Architecting for evolvable systems implies a complex decision-making process in which multiple aspects need to be taken into consideration, e.g., stakeholders' needs and goals, multiple quality requirements with competing priorities, various architectural solutions with divergent implications on quality requirements.

As a software system is subject to a substantial amount of evolutionary changes, e.g. software technology changes, system migration to product line architecture, an evolvable software system can often reflect these changes to adequately fulfill its roles and remain relevant to stakeholders. As stated in Chapter 1, the main objective of our research is to improve the capability to understand and analyze systematically the evolution of a software architecture. Concretely, we address the following research questions:

- What software characteristics are necessary to constitute an evolvable software system?
- How to assess evolvability of a software system in a systematic manner when evolving the architecture to embrace potential architectural requirements caused by a certain change, e.g., ever-changing business requirements, advances of technology?

In Chapter 3, we have presented the wide spectrum of approaches that cover five main categories of themes and aim to support software architecture evolution. In this chapter, we will describe software architecture evolution characterization, and propose an architecture evolvability analysis process that provides repeatable techniques for performing the activities to understand and support software architecture evolution. The evolvability

analysis process addresses various aspects, i.e., quality consideration during design, architectural quality evaluation, economic valuation, and architectural knowledge management. The repeatable techniques include:

- A structured qualitative method for analyzing evolvability at the architectural level;
- A quantitative evolvability analysis method with explicit and quantitative treatment of stakeholders' evolvability concerns and potential architectural solutions' impacts on evolvability.

These techniques as well as the software evolvability model are inspired from the comprehensive literature surveys and our experiences in working with various industrial software systems. They have been refined and validated in practice (see Chapter 5). This chapter focuses on the introduction of the software evolvability model and evolvability analysis processes.

The remainder of the chapter is structured as follows. Chapter 4.1 describes the software evolvability model which is the basis for evolvability analysis process. Chapter 4.2 presents the general software evolvability analysis process along with detailed descriptions of the qualitative and quantitative architecture evolvability analysis methods.

## 4.1 Software Evolvability Model

Rowe and Leaney [153] state that software evolvability is a multifaceted quality attribute. Based on survey of literatures, e.g., the definition by Rowe and Leaney [153], the analysis of the software quality challenges and assessment by Fitzpatrick et al. [67], the types of change stimuli and evolution [50], and the taxonomy of software change based on various dimensions that characterize or influence the mechanisms of change [39], we have found that only having a collection of the subcharacteristics of maintainability as defined in the ISO software quality standard [89] is not sufficient for a software system to be evolvable. Therefore, we have complimented and identified subcharacteristics that are of primary importance for an evolvable software system, and outlined a software evolvability model that provides a basis for analyzing and evaluating software evolvability. The proposed model structure is inspired by ISO 9126 quality model [89], which describes complex quality criteria through breaking them down into concrete subcharacteristics. Besides, we have also looked into other quality models, and identified subcharacteristics related to evolvability (see Chapter 2).

The overall model structure and its basic principles are shown on Figure 4-1. The evolvability model refines software evolvability into a collection of subcharacteristics that can be measured through a number of corresponding measuring attributes. The idea with the evolvability model is to further derive the identified subcharacteristics to the extent when we are able to quantify them by defining metrics to measure relevant measuring attributes for each subcharacteristic, and/or make appropriate reasoning about the quality of service (QoS) for subcharacteristics that are difficult to be quantified (e.g., architectural integrity, described below).

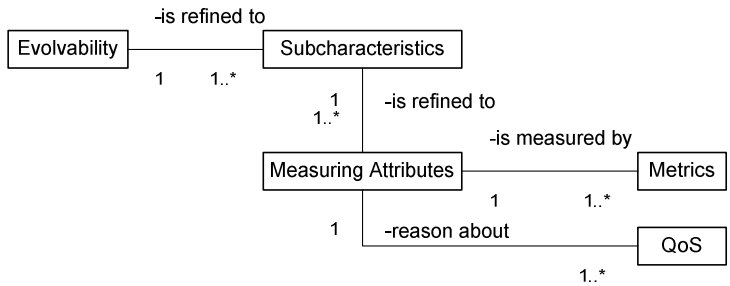


**Figure 4-1: Software evolvability model**

The model and its validation are based on industrial requirements of long-life software-intensive systems within different domains. In particular, they are the results from case studies [31] [33, 34], and are valid for a class of long-lived industrial software-intensive systems that often are exposed to many, and in most cases evolutionary changes. For these types of systems we have identified the following evolvability subcharacteristics, with examples of measuring attributes for each subcharacteristic:

- *Analyzability* describes the capability of the software system to enable the identification of influenced parts due to change stimuli; its measuring attributes include modularity, complexity, and architectural documentation.
- *Architectural Integrity* describes the non-occurrence of improper alteration of architectural information; its measuring attributes include architectural documentation.
- *Changeability* describes the capability of the software system to enable a specified modification to be implemented and avoid unexpected effects; its measuring attributes include complexity, coupling, change impact, encapsulation, reuse, and modularity.

- *Extensibility* describes the capability of the software system to enable the implementations of extensions to expand or enhance the system with new features; its measuring attributes include modularity, coupling, encapsulation, and change impact.
- *Portability* describes the capability of the software system to be transferred from one environment to another; its measuring attributes include mechanisms facilitating adaptation to different environments.
- *Testability* describes the capability of the software system to validate the modified software; its measuring attributes include complexity, and modularity.
- *Domain-specific attributes* are the additional quality subcharacteristics that are required by specific domains.

These evolvability subcharacteristics are the main enablers of evolvability. However, we do not exclude the possibilities that other domains might have slightly different set of subcharacteristics, in particularly with domain-specific attributes. For instance, the World Wide Web domain requires additional quality characteristics such as visibility, intelligibility, credibility, engagibility and differentiation [67]. Component exchangeability in the context of service reuse is another example within the distributed domain, e.g., wireless computing, component-based and service-oriented applications.

According to Parnas [137], software evolution is very often negatively influenced by architectural drift, feature creep, and progressive hardware dependence. However, with the identified evolvability subcharacteristics in mind, we have a basis upon which different systems can be examined with respect to evolvability. Any system design and architectural decisions that do not explicitly address one or more of these subcharacteristics probably will undermine the system's ability to be evolved. Therefore, the software evolvability model is a way to articulate subcharacteristics for an evolvable system that an architecture must support. It is established as a first step towards analyzing and quantifying evolvability, a base and checkpoints for evolvability evaluation and improvement, and is an integral part in qualitative and quantitative evolvability analysis processes (see Chapter 4.2).

## 4.2 Software Architecture Evolvability Analysis Process

The software architecture evolvability analysis process (AREA) engages stakeholders throughout the system development and evolution lifecycle to discover the driving architectural requirements, stakeholders' evolvability concerns, and potential architectural solutions' impact on evolvability of a software system. It can be carried out at many points during a system's lifecycle, e.g., during the design phase to evaluate prospective candidate designs, validating the architecture before further commencement of development, or evaluating architecture of a legacy system that is undergoing modification, extension, or other significant upgrades.

The analysis process is stakeholder-focused, and therefore, is dependent on the participation of involved stakeholders of various roles, such as architects, development team, research team, project leader, and product managers. All participants are encouraged to comment and state their opinions during the workshops and interviews. Consequently, the analysis process provides increased stakeholder communication.

The evolvability analysis can be conducted by an internal assessment team or an external evaluation team. Having an internal assessment team requires discipline as it tends to be subject to more bias and influence, especially if the team is part of the organization that is responsible for evolving the architecture. Having an external assessment team is less affected by biased opinions, though with a weakness of its lack of knowledge of the system in focus.

The results of the evolvability analysis process include:

- Prioritized architectural requirements;
- Stakeholders' evolvability concerns;
- Candidate architectural solutions;
- Architectural solutions' impact on evolvability.

It is a challenging task for an architect to choose among competing candidate architectural solutions and ensure that the system constructed from the architecture satisfies its stakeholders' needs. Nevertheless, the results from the software evolvability analysis process provide useful input to an architect to design and evolve the architecture.

As stated in the beginning of this chapter, the techniques embedded in the software evolvability analysis process include a qualitative evolvability

analysis method which will be introduced in Chapter 4.3, and a quantitative evolvability analysis method which will be introduced in Chapter 4.4. Both the qualitative and quantitative evolvability analysis methods that we have developed share commonalities at the conceptual level. This is reflected in the general architecture evolvability analysis process (AREA), which is illustrated in Figure 4-2.

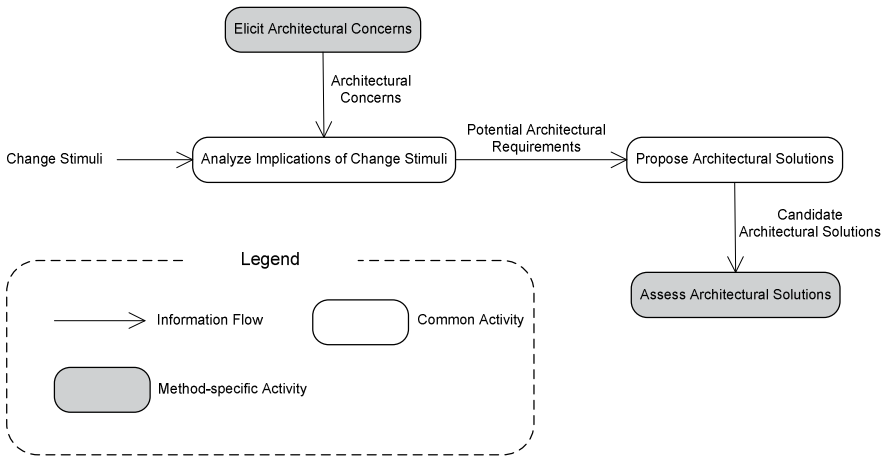


**Figure 4-2: Software architecture evolvability analysis process (AREA)**

A change stimulus is the event that causes the architecture to respond or change, and is therefore the driver for an architecture evolvability analysis process. A change stimulus can be a concrete functional requirement from customers, a goal for the development organization (e.g., to be more productive in the numbers of features developed in each product release), or an adaption to future technology trends. Based on our experiences in industry, it is often the software architecture core team (system architects), product manager, and product/system owner who synchronize and reach a consensus on the choice of change stimulus. Nevertheless, it happens also that a shifted business goal from the higher management level (e.g., to enable geographically distributed development) determines the change stimulus to focus on. Because the analysis on where the change stimuli come from and which one to opt for is a research topic by itself, we focus in this dissertation the subsequent evolvability analysis process once a change stimulus has been decided.

The analysis of the implications of a change stimulus needs to take into account the architectural concerns among stakeholders before articulating

potential architectural requirements. Based on the identified potential architectural requirements, a collection of candidate architectural solutions are proposed to address these requirements, and thereafter, are assessed against evolvability subcharacteristics. The assessment of candidate architectural solutions ensures that the choice of an architectural solution is well analyzed instead of relying on intuition.

The related artifacts in the software evolvability analysis process include:

- **Change stimuli**

A stimulus is a change condition that needs to be considered from architectural perspective. Change stimuli trigger an initiation of the architecture evolvability analysis process. A change stimulus can be a concrete change, a future change that we know will happen, or a change that we currently have no idea of, but belonging to a particular class of change related to environment, organization, process, technology and stakeholders' needs. These change stimuli have impact on the software system in terms of software architecture evolution and embedded quality attributes. A change stimulus results in a collection of potential requirements that the software architecture needs to adapt to.

- **Stakeholders' concerns**

The IEEE 1471 standard [88] defines architectural concerns as "*interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns include system considerations such as performance, reliability, and evolvability*". In this dissertation, we focus on evolvability concerns.

- **Potential architectural requirements**

Potential architectural requirements are requirements that influence software architecture and are essential for accommodating change stimuli.

- **Candidate architectural solutions**

Candidate architectural solutions are alternative solutions that reflect design decisions. The description of an architectural solution may include following information:

  - *Problem description* which describes the problem and disadvantages of the original design of the architecture or fragment of the architecture;

- *Requirement* which refers to the new requirements that the architecture needs to fulfill;
- *Improvement solution* which is the architectural solution to design problems;
- *Rationale and architectural consequence* which describes the rationale of the solution proposal and architectural implications of the candidate solution on quality attributes;
- *Estimated workload* for implementation and verification.

The main activities in the software evolvability analysis process include:

- **Elicit architectural concerns**

This activity extracts architectural concerns with respect to evolvability subcharacteristics among stakeholders either qualitatively or quantitatively.

- *Qualitative elicitation*

Architecture workshops are conducted so that the stakeholders discuss and identify potential architectural requirements that are thereof mapped against the evolvability subcharacteristics. Thus the identified architectural requirements and their prioritization reflect stakeholders' architectural concerns with respect to evolvability subcharacteristics.

- *Quantitative elicitation*

Individual interviews with respective stakeholders are conducted so that stakeholders representing different roles provide their views and preferences on evolvability subcharacteristics by a pair-wise comparison of subcharacteristics with respect to their relative importance. Thus the preference weights on evolvability subcharacteristics from a stakeholder's perspective are quantified.

- **Analyze implications of change stimuli**

This activity analyzes the architecture for evolution, and identifies the impact of change stimuli on the current architecture. Accordingly, this activity focuses on defining the problems that the architecture needs to solve, and examines change stimuli and architectural concerns in order to obtain a set of potential architectural requirements.

- **Propose architectural solutions**

This activity focuses on proposing architectural solutions that accommodate to a set of potential architectural requirements.

- **Assess architectural solutions**

This activity ensures that the architectural design decisions made are appropriate for software architecture evolution. The candidate architectural solutions are assessed against evolvability subcharacteristics, i.e., the implications of the potential architectural strategies and evolution path of the software architecture are analyzed either qualitatively or quantitatively.

- *Qualitative assessment* in which the determination of potential architectural solutions is made on a qualitative level in terms of their impact (i.e., positive or negative) on evolvability subcharacteristics.
- *Quantitative assessment* in which the judgment on how well each candidate architectural solution supports different evolvability subcharacteristics is quantified.

As we see from the general software evolvability analysis process, the basic architecting activities such as analyzing implications of change stimuli and proposing architectural solutions are the same for both the qualitative and quantitative evolvability analysis. The major variation can be observed in the different details with respect to stakeholders' architectural concerns elicitation and assessment of architectural solutions against evolvability subcharacteristics. The following two chapters will concretize the steps performed in the qualitative and quantitative evolvability analysis process respectively.

## 4.3 Qualitative Evolvability Analysis Method

The qualitative evolvability analysis method [34] starts with identification of change stimuli's implications, guides architects through the analysis of potential architectural requirements that the software architecture needs to adapt to, and continues with identification of potential architecture refactoring solutions along with their implications. Through the analysis process, the implications of the potential improvement proposals and evolution path of the software architecture are analyzed with respect to evolvability subcharacteristics.

The qualitative architecture evolvability analysis method, as shown in Figure 4-3, is divided into three main phases.

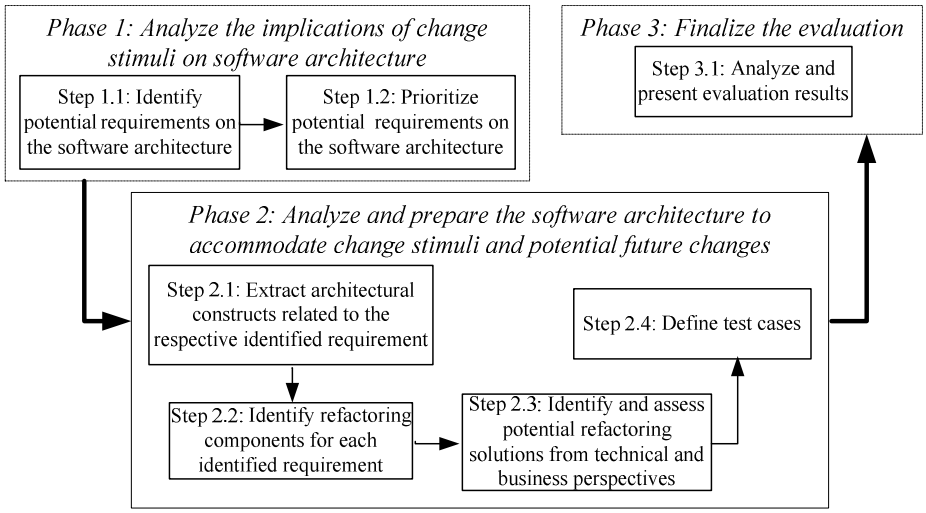


**Figure 4-3: Qualitative architecture evolvability analysis method**

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

This phase analyzes the architecture for evolution, and identifies the impact of change stimuli on the current architecture. Software evolvability concerns (1) business, and (2) technical issues [31], since the stimuli of changes come from both perspectives, e.g., environment, organization, process, technology and stakeholders' needs. These change stimuli have impact on the software system in terms of software structures and/or functionality. This phase includes two steps:

- **Step 1.1**: Identify requirements on the software architecture.

  Any change stimulus (see Chapter 4.2) results in a collection of requirements that the software architecture needs to adapt to. The aim of this step is to extract requirements that are essential for software architecture enhancement so as to cost-effectively accommodate to change stimuli. Architecture workshops are conducted, where the stakeholders discuss and identify architecture requirements. Afterwards, the identified requirements are checked against the evolvability subcharacteristics in order to ensure the completeness of the identified requirements.

- **Step 1.2**: Prioritize requirements on the software architecture.

  In order to establish a basis for common understanding of the architecture requirements among stakeholders within the organization,

all the requirements identified from the previous step need to be prioritized.

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

This phase focuses on the identification and improvement of the components that need to be refactored. It includes four steps:

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

In this step, we mainly focus on identifying architectural constructs (i.e., subsystems and components) that are related to each identified requirement.

- **Step 2.2**: Identify refactoring architectural components for each identified issue.

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

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

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

- **Step 2.4**: Define test cases.

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

**Phase 3**: Finalize the evaluation.

In this phase, the previous results are incorporated and structured into a collection of documents. This phase includes one step.

- **Step 3.1**: Present evaluation results.

The evaluation results include:

- Identified and prioritized requirements on the software architecture;
- Identified components/modules that need to be refactored for enhancement or adaptation;
- Refactoring investigation documentation which describes the current situation and solutions to each identified candidate component that need to be refactored, including estimated workload;
- Test scenarios.

## 4.4 Quantitative Evolvability Analysis Method

As architecture is influenced by stakeholders representing different concerns and goals, the business and technical decisions that articulate the architecture tend to exhibit tradeoffs, and need to be negotiated and resolved. In circumstances when there is a lack of a common-shared view on prioritizations of evolvability subcharacteristics among stakeholders, to avoid intuitive choice of architectural solutions, the quantitative evolvability analysis method [32] is used to enable explicit and quantitative treatment of stakeholders' prioritization of evolvability subcharacteristics and their preferences on design solutions.

The quantitative approach is based on the qualitative method, and focuses on two constituent steps of the qualitative evolvability analysis method in which tradeoff analysis is concerned, i.e., step 1.2 in phase 1 (Prioritize requirements on the software architecture), and step 2.3 in phase 2 (Identify and assess potential refactoring solutions). These two steps entail subjective judgments with regard to stakeholders' preferences of evolvability subcharacteristics, as well as their choices of architectural solutions. These subjective judgments constitute accordingly a multiple-attribute decision making process when architecting for evolvable software systems, as illustrated in Figure 4-4. The stakeholders' preferences on evolvability subcharacteristics are determined by their different viewpoints, and the choice of architectural alternatives exhibits their respective impacts on evolvability. Meanwhile, the choice of architectural solution is constrained by stakeholders' preference information on evolvability subcharacteristics.

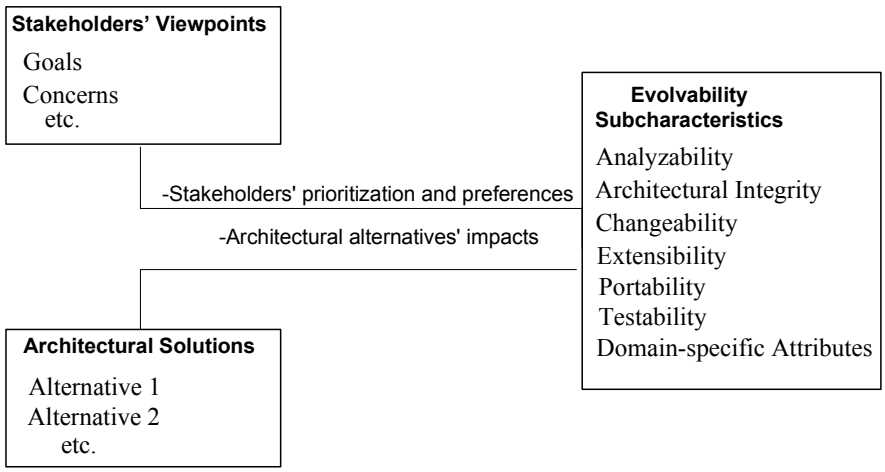


**Figure 4-4: Multiple-attribute decision making process**

The quantitative evolvability analysis method provides a structured way in

- Understanding subjective decision making process by quantitatively eliciting stakeholders' preferences for desired evolvability subcharacteristics;
- Obtaining quantitative understanding of the impacts of architectural solutions on evolvability.

Through the relative importance measuring process, we gain an explicit view on how stakeholders prioritize numerous evolvability subcharacteristics, as well as the rationale behind a choice of an architectural alternative. Thus, the quantitative approach provides decision support and helps to avoid intuitive prioritization of evolvability subcharacteristics and intuitive choice of architectural solutions.

In Chapter 4.4.1, we present briefly the *Analytic Hierarchy Process (AHP)* [156] method, upon which the quantitative evolvability analysis method is based. Then we will describe the steps involved in the quantitative evolvability analysis method in Chapter 4.4.2.

### 4.4.1 Analytic Hierarchy Process

To obtain quantitative data with regard to stakeholders' preferences on evolvability subcharacteristics and architectural alternatives' impacts on evolvability, we use Analytic Hierarchy Process (AHP) [156], because it is a multiple-attribute decision making method that enables quantification of

subjective judgments. It makes relative assessments through pair-wise variable comparison, and consists of five basic steps:

- **Step 1**: Create an *n* x *n* matrix *N*, in which *n* is the number of variables to be compared.
- **Step 2**: Perform pair-wise comparison of the variables with respect to importance. Populate the matrix with the comparison values ($n_{ij}$ for comparison value between i-th and j-th variable). The interpretation of the scales for comparison is shown in Table 4-1.

**Table 4-1: Scale for pair-wise comparison**

| Scale | Explanation |
|---|---|
| 1 | Variable *i* and *j* are of equal importance |
| 3 | Variable *i* is slightly more important than *j* |
| 5 | Variable *i* is highly more important than *j* |
| 7 | Variable *i* is very highly more important than *j* |
| 9 | Variable *i* is extremely more important than *j* |
| 2,4,6,8 | Intermediate values for compromising between the other numbers |

- **Step 3**: Compute eigenvector of the *n* x *n* matrix. We apply the *averaging over normalized columns* method [156] which uses the following equations:

a) Calculate sum of the columns;

$$n_j = \sum_{i=1}^{n} (n_{ij})$$

b) Divide each element in a column by the sum of the column, resulting in a new matrix *M* with elements $m_{ij}$;

$$m_{ij} = n_{ij}/n_j$$

c) Calculate sum of each row in the new matrix;

$$m_i = \sum_{j=1}^{n} (m_{ij})$$

d) Normalize the sum of rows to obtain priority vector P (with elements $P_i$) by dividing by *n*, which is the number of variables.

$$P_i = m_i/n \quad (i \text{ is an integral and } 1 \leq i \leq n)$$

- **Step 4**: Assign a relative importance to the variables, each accounts for a certain amount of percentage of the importance of the variables. The first variable is assigned the first element in the priority vector; the second variable is assigned the second element in the priority vector, and so on.
- **Step 5**: Evaluate consistency of subjective judgment. The consistency ratio is computed in two steps:
  - Consistency index *CI* is computed as $CI = (\lambda max - n) / (n - 1)$, in which $\lambda max$ is the maximum principle eigen value of the $n$ x $n$ matrix.
  - Consistency ratio *CR* is computed as $CR = CI / RI$, in which *RI* is the random index. The random index is generated to take into account randomness, and is used to normalize the consistency index. The standard *RI* value can be obtained from [156] as shown in Table 4-2, in which the first row shows the order of the matrix (i.e., the value of $n$), and the second row shows the corresponding *RI* value. The smaller *CR* value is, the more consistent is the obtained comparison. According to [156], an approximate size of the expected consistency ratio of 0.10 or less is considered acceptable, though, in practice, higher values are often obtained.

**Table 4-2: Random index values**

| **Order of matrix** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Random Index** | 0 | 0 | 0.58 | 0.90 | 1.12 | 1.24 | 1.32 | 1.41 | 1.45 | 1.45 |

## 4.4.2 Quantitative Evolvability Analysis Method

The quantitative architecture evolvability analysis method extends the qualitative method with quantitative information that is needed for choosing among architectural solutions, and is divided into three main phases:

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

This phase elicits the architectural concerns among stakeholders and analyzes the architecture for evolution in order to accommodate change stimuli.

- **Step 1.1**: Elicit stakeholders' views on evolvability subcharacteristics.

In this step, individual interviews are conducted with respective stakeholders in order to elicit their views on evolvability subcharacteristics. Domain-specific attributes are identified as well. In addition, the interpretation of evolvability subcharacteristics in the specific context is discussed.

- **Step 1.2**: Extract stakeholders' prioritization and preferences of evolvability subcharacteristics.

In this step, stakeholders representing different roles provide their preferences on evolvability subcharacteristics by a pair-wise comparison of subcharacteristics $(Q_i, Q_j)$ with respect to their relative importance. The AHP weighting scale shown in Table 4-1 is used to determine the relative importance for each evolvability subcharacteristic pair. Note that the domain-specific attributes might comprise several additional quality characteristics that are required by a specific domain. Therefore, each of these domain-specific quality attributes is also included for pair-wise comparison together with the other evolvability subcharacteristics. The pair-wise comparison is conducted for all pairs, hence, $n(n\text{-}1)/2$ comparisons are made by each stakeholder. Afterwards, for each stakeholder, the equations in AHP method (see Chapter 4.4.1) are used to create a priority vector signifying the relative preference of evolvability subcharacteristics. As different stakeholder roles might have diversified preferences on evolvability subcharacteristics, for each evolvability subcharacteristic, we obtain normalized preference on an evolvability subcharacteristic by dividing sum of the preference of each stakeholder role by the number of roles.

The description below concretizes the calculation procedure, describing the calculation of preferences of subcharacteristics aggregated from stakeholders' perspectives.

A matrix of pair-wise comparison is shown below, in which $S_1$ represents one stakeholder role, $Q_1, Q_2$ and $Q_k$ are evolvability subcharacteristics, $I_{ij}$ represents pair-wise comparison in terms of relative importance based on Table 4-1 (Note $I_{ij} = 1$ if $i = j$).

| $S_1$ | $Q_1$ | $Q_2$ | … | $Q_k$ |
|---|---|---|---|---|
| $Q_1$ | $I_{11}$ | $I_{12}$ | | $I_{1k}$ |
| $Q_2$ | $I_{21}$ | $I_{22}$ | | $I_{2k}$ |
| ⋮ | | | | |
| $Q_k$ | $I_{k1}$ | $I_{k2}$ | | $I_{kk}$ |

Then by applying equation a), we get the sum of the columns:

$$n_j = \sum_{i=1}^{k} (n_{ij})$$

By applying equation b), we get the following new matrix:

$$m_{ij} = n_{ij}/n_j$$

Then by applying equations c) and d), we get normalized preference weight information of subcharacteristic $Q_i$ from stakeholder $S_1$ perspective as shown in equation (1) below:

$$PQ_{is1} = \frac{\sum_{j=1}^{k} (m_{ij})}{k} \quad (i \text{ is an integral and } 1 \leq i \leq k) \qquad \textbf{(1)}$$

Likewise, the values indicating the preference weights of subcharacteristics ($Q_1, Q_2, \ldots Q_k$) from stakeholder $S_2$ perspective are calculated. We designate them as $PQ_{1s2}, PQ_{2s2}, \ldots PQ_{ks2}$. The same pattern applies to all the other stakeholder roles.

Given that the preference consistency is correct, the overall stakeholders' preference weight on subcharacteristic $Q_i$ is calculated by aggregating the preferences from *n* number of stakeholders as shown in equation (2) below:

$$PQ_i = \frac{\sum_{j=1}^{n} (PQ_{isj})}{n} \quad (i \text{ is an integral and } 1 \leq i \leq n) \qquad \textbf{(2)}$$

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

This phase focuses on the identification and evaluation of candidate architectural solutions to accommodate change stimuli. The stakeholders' preferences of evolvability subcharacteristics from the previous phase are used in the last step in this phase to calculate the candidate architectural solutions' overall impacts on software evolvability.

- **Step 2.1**: Identify candidate architectural solutions.

In this step, candidate architectural solutions are identified along with their benefits and drawbacks.

- **Step 2.2**: Assess candidate architectural solutions' impacts on evolvability subcharacteristics.

In this step, system architects or main technical responsible persons provide their judgment on how well each architectural alternative supports different evolvability subcharacteristics. This is firstly done by a pair-wise comparison of the architectural alternatives $(Alt_i, Alt_j)$ with respect to a certain evolvability subcharacteristic, using the weighting scale in Table 4-1. Next, for each evolvability subcharacteristic, the aforementioned equations in AHP method (see Chapter 4.4.1) are used to create a priority vector signifying the relative weight of how well different architectural alternatives support a specific evolvability subcharacteristic. Afterwards, recalling the overall weights, i.e., stakeholders' preference weight of evolvability subcharacteristics and the weight of how well different architectural alternatives support a specific evolvability subcharacteristic, we can obtain a normalized value, designating the overall weight for each architectural alternative's support on evolvability in general.

The following description concretizes the calculation procedure, describing the calculation of architectural alternatives' overall support on software evolvability.

A matrix of pair-wise comparison is shown below, in which $Q_1$ represents one of the evolvability subcharacteristics, $Alt_1$, $Alt_2$ and $Alt_k$ are architectural alternatives, $S_{ij}$ represents pair-wise comparison (based on Table 4-1, $S_{ij} = 1$ if $i = j$) in terms of relative support of each alternative on a certain subcharacteristic such as $Q_1$ as shown below:

| $\boldsymbol{Q_1}$ | $\boldsymbol{Alt_1}$ | $\boldsymbol{Alt_2}$ | ⋯ | $\boldsymbol{Alt_k}$ |
|---|---|---|---|---|
| $Alt_1$ | $S_{11}$ | $S_{12}$ | | $S_{1k}$ |
| $Alt_2$ | $S_{21}$ | $S_{22}$ | | $S_{2k}$ |
| ⋮ | | | | |
| $Alt_k$ | $S_{k1}$ | $S_{k2}$ | | $S_{kk}$ |

Then by applying equation a), we get the sum of the columns:

$$n_j = \sum_{i=1}^{k} (n_{ij})$$

By applying equation b), we get the following new matrix:

$$m_{ij} = n_{ij}/n_j$$

Then by applying equations c) and d), we get normalized support rates of the respective architectural alternative with respect to $Q_1$ as shown below, in which $PAlt_{iq1}$ indicates impact of the alternative $Alt_i$ on subcharacteristic $Q_1$, i.e., how well each architectural alternative supports $Q_1$.

$$PAlt_{iq1} = \frac{\sum_{j=1}^{k} (m_{ij})}{k} \quad (i \text{ is an integral and } 1 \leq i \leq k)$$

Likewise, the values indicating how well the alternatives support other subcharacteristics ( $Q_2 \ldots Q_k$ ) are calculated in the same pattern.

- **Step 2.3**: Assess candidate architectural solutions' overall impacts on software evolvability

Given that the judgment of architectural alternatives' support on subcharacteristics is consistent, the overall weights of an alternative' support on evolvability is calculated by aggregating the preferences of subcharacteristics from the previous quantitative analysis (i.e., $PQ_1$, $PQ_2$ … $PQ_k$ ) as expressed in equation (3) below:

$$W_{Altm} = \sum_{i=1}^{k} \left( PQ_i \times PAlt_{mqi} \right) \qquad (m \text{ is an integral and } 1 \leq m \leq k) \tag{3}$$

**Phase 3:** Finalize the evaluation.

In this phase, the previous results are incorporated and summarized. This phase includes one step.

- **Step 3.1**: Present evaluation results.

The evaluation results include:

- Identified evolvability subcharacteristics including domain-specific attributes;
- Quantified prioritization of evolvability subcharacteristics by respective stakeholders involved;
- Common understanding of the contexts of evolvability subcharacteristics;
- Identified architectural solution candidates to cope with change stimuli;
- Quantified prioritization of the impacts of each architectural candidate on evolvability subcharacteristics.

## 4.5 Characterization of the Qualitative and Quantitative Methods

Based on our experiences in applying the evolvability analysis processes (to be described in Chapter 5), both the qualitative and quantitative analysis methods can be used as an integral part of the software development and evolution process to assess software architectures for evolution. They share the common themes of

- Systematically addressing quality requirements driven by change stimuli; and
- Assisting architects in analyzing the impact of potential architectural solutions on evolvability subcharacteristics before determining the potential evolution path for the software architecture.

There are also variations between the two methods as detailed in the following sections (summarized in Table 4-3).

**Table 4-3: Characterizations of the qualitative and quantitative evolvability analysis methods**

| | Application Contexts | Approaches Used | Analysis Output |
|---|---|---|---|
| **Qualitative Analysis** | Common perception on important quality attributes and their prioritization within organization.<br>A preferred architectural solution can be decided based on the qualitative impact data. | Architecture workshops with all involved stakeholders to discuss prioritization of potential architectural requirements.<br>Architecture workshops with architects to discuss architectural solutions and qualitative impacts on evolvability. | Identified and prioritized potential architectural requirements.<br>Qualitative analysis of architectural solutions' impact on evolvability subcharacteristics. |
| **Quantitative Analysis** | Numerous stakeholder roles representing different concerns.<br>Unclear perception and prioritization of important quality attributes.<br>Difficult to decide the preferred architectural solution based on qualitative data. | Interviews with individual stakeholders to discuss preferences of evolvability subcharacteristics.<br>Architecture workshops with architects to discuss architectural solutions and quantitative impacts on evolvability.<br>Analytic Hierarchical Process method. | Quantified stakeholders' preferences on evolvability subcharacteristics.<br>Quantified prioritization of candidate architectural solutions' impacts on evolvability. |

### 4.5.1 Application Contexts

The application contexts for using the qualitative and/or quantitative evolvability analysis methods are concerned with two aspects:

- **Stakeholders' perception on quality attributes**

Software architecture is influenced by system stakeholders [18]. In circumstances when there are numerous roles of stakeholders, representing different and sometimes contradictory concerns and goals, an explicit quantitative assessment of stakeholders' preferences on evolvability subcharacteristics will strengthen qualitative data, and assist architects in making architectural design decisions, especially when there is not a clear view within an organization on important quality attributes and their prioritization.

- **Candidate architectural solutions' impacts on evolvability**

Architects must often make architectural design decisions, and give preference to a certain architectural solution. In circumstances when there are multiple architectural alternatives to choose among, each of which exhibiting divergent impacts on evolvability subcharacteristics, a quantitative assessment of candidate architectural solutions' impacts on evolvability subcharacteristics will guide and support architects to avoid making intuitive decisions in software architecture evolution, especially when the qualitative data is not sufficient for determining a preferred candidate architectural solution.

### 4.5.2 Approaches Used in the Analysis Process

The qualitative evolvability analysis is mainly conducted through:

- Architecture workshops in which all involved stakeholders participate to identify and prioritize potential architectural requirements;
- Architecture workshops in which the architects discuss potential architectural solutions along with their qualitative impacts on evolvability.

The quantitative evolvability analysis is based on AHP [156], and conducted through:

- Interviews with respective stakeholder to extract individual stakeholder's preference on evolvability subcharacteristics;
- Architecture workshops in which the architects discuss potential architectural solutions along with their quantitative impacts on evolvability.

### 4.5.3 Analysis Output

The main output of the qualitative evolvability analysis method includes the identified and prioritized potential architectural requirements, identified components that need to be refactored, candidate architectural solutions along with their qualitative evolvability impact analysis data, as well as test scenarios.

The main output of the quantitative evolvability analysis method includes quantified prioritization of evolvability subcharacteristics among stakeholders, and identified candidate architectural solutions along with their quantitative evolvability impact data.

### 4.5.4 Choosing Between Qualitative and Quantitative Methods

The choice of which analysis method to use is based on the specific application contexts and the expected analysis output. The following questions are related to application contexts, and can be used as checkpoints (with Yes or No answer) for determining when to use which analysis method:

- Are there numerous roles of stakeholders with divergent concerns and goals? (Y/N)
- Is it clear within the software development organization regarding the prioritizations of the important quality attributes that concern the evolution of the system in focus? (Y/N)
- Is it difficult to determine a preferred candidate architectural solution among the multiple architectural alternatives due to their various impacts on evolvability subcharacteristics? (Y/N)

The decision diagram for choosing appropriate analysis method based on the answers to the questions is shown in Figure 4-5. The first two checkpoints concern the stakeholders' perception on quality attributes, and are related to the first phase of both qualitative and quantitative analysis. The third checkpoint concerns selecting a preferred architectural solution, and relates to the second phase of the two methods. It is therefore possible to combine the qualitative and quantitative analysis methods, e.g., starting with a qualitative analysis and complement with quantitative data, or vice versa. Depending on the answers to the questions, the corresponding phase of either the qualitative or quantitative analysis can be selected.

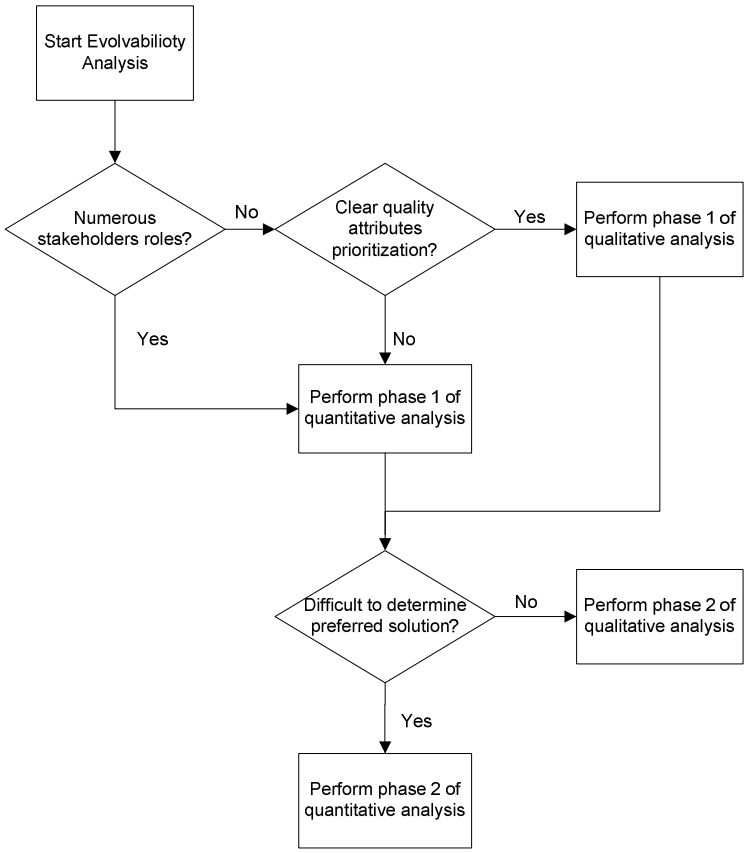


**Figure 4-5: A decision diagram for choosing between the qualitative and quantitative methods**

## 4.6 Summary

Motivated by the need to understand software architecture evolution and to investigate ways to analyze software evolvability to support this evolution, the central theme of this chapter focuses on two particular aspects:

- Identify software characteristics that are necessary to constitute an evolvable software system

In this aspect, we have defined a software evolvability model, which refines evolvability into a collection of subcharacteristics. This model is established as a first step towards analyzing and quantifying software

evolvability. It provides a basis for analyzing software evolvability, and a check point for evolvability evaluation and improvement.

- Assess evolvability in a systematic manner

In this aspect, we have proposed and described the software architecture evolvability analysis process (AREA) which is based on the evolvability model. The AREA process provides also repeatable techniques for supporting software architecture evolution:

- Qualitative architecture evolvability analysis method that focuses on improving the capability of being able to understand and analyze systematically the impact of change stimuli on software architecture evolution.
- Quantitative architecture evolvability analysis method that provides quantifications of stakeholders' evolvability concerns and potential architectural solutions' impacts on evolvability.

We have now introduced the software evolvability model and the evolvability analysis process, with in-depth description of the steps in both the qualitative and quantitative evolvability analysis methods. In next chapter, we will present the industrial case studies of using the evolvability model and analysis processes as they were realized in real evaluations.

# Chapter 5. Analyzing Proprietary Systems

In Chapter 4 we have introduced the qualitative and quantitative evolvability assessments manifested in the software evolvability analysis process. This chapter describes the validations, i.e., their applications in two large-scale industrial software systems at ABB and Ericsson. The experiences and reflections in the case studies with respect to managing software architecture evolution guided by the evolvability analysis at architectural level are described as well.

## 5.1 Case Study I. Qualitative Software Evolvability Analysis

This section describes the case study in which we applied the qualitative software evolvability analysis method. The system that we investigated is an automation control system at ABB.

### 5.1.1 Context of the Case Study

The case study was based on a large automation control system at ABB. During the long history of product development, several generations of automation controllers have been developed as well as a family of software products, ranging from programming tools to varieties of application software. The case study focused on the latest generation of the robot controller.

The robot controller software consists of more than three million lines of code written in C/C++, and uses a complex threading model, with support for a variety of different applications and devices. It has grown in size and complexity as new features and solutions have been added to enhance functionality and to support new hardware, such as devices, I/O boards and production equipment. Such a complex system is challenging to evolve. Due to different measures such as organizational and lifecycle process

improvements, the system keeps maintainability, but the evolvability becomes more difficult since the increased complexity in turn leads to decreased flexibility, resulting in problems to add new features. Consequently, it would become costly to adapt to new market demands and penetrate new markets.

Our particular system was delivered as a single monolithic software package, which consists of various software applications developed by distributed development teams. These applications aim for specific tasks in painting, welding, gluing, machine tending and palletizing, etc. In order to keep the integration and delivery process efficient, the initial architectural decision was to keep the deployment artifact monolithic; The complete set of functionality and services was present in every product even though not everything was required in the specific product. As the system grew, it became more difficult to ensure that the modifications of specific application software would not affect the quality of other parts of the software system.

The original coarse-grained architecture of the controller is depicted in Figure 5-1. The lower layer provides an interface to the upper layer, and allows the source code of the upper layer to be compiled and used on different hardware platforms and operating systems. The complete set of interdependencies between subsystems within each layer is not captured in the figure.

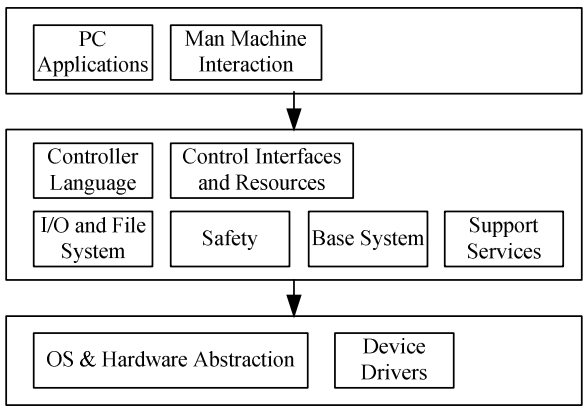


**Figure 5-1: A conceptual view of the original software architecture**

The main problem with this software architecture was the existence of tight coupling among some components that reside in different layers. This led to additional work required at a lower level to modify some existing functionality and add support for new functionality in various applications. For instance, the system is required to perform certain tasks during start-up

and shutdown in the controller. Some routines for handling such tasks had to be hard-coded, i.e., the application developers had to edit in the source code of e.g., *Support Services* subsystem in the lower layer, which was developed by another group of developers. Accordingly, source code updates had to be done not only on the application level, but through several layers, several subsystems and components. Recompilation of the whole code base was required. This means that application developers need to have thorough knowledge of the complete source code. It also constituted a bottleneck in the effort to enable distributed application development. To continue exploiting the substantial software investment made and to continuously improve the system for longer productive lifetime, it became essential to explicitly address evolvability.

We want to emphasize here that the problem raised is not a problem of maintainability. The major problems arose when brand new (very different) features or different development paradigms, shifting business and organizational goals were introduced, so the problems were related to software evolvability.

### 5.1.2 Evolvability Subcharacteristics from Case Perspective

We explain below each evolvability subcharacteristic in conjunction with the case study context:

- **Analyzability** The release frequency of the controller software was twice a year, with around 40 various new major requirements that needed to be implemented in each release. These requirements may have impact on different attributes of the system, and the possible impact must be analyzed before the implementation of the requirements.
- **Architectural Integrity** A strategy for communicating architectural decisions that we found out from the case study was to appoint members of the core architecture team as technical leaders in respective development projects. However, this strategy, although helpful to certain extent, did not completely prevent developers from insufficient understanding and/or misunderstanding of the initial architectural decisions, resulting in unconscious violation of architectural conformance.
- **Changeability** Due to the monolithic characteristic of the controller software, modifications in certain parts of the software package led to some ripple effects, and required recompiling, reintegrating and retesting of the whole system. This led to inflexibility of patching, and customers

had to wait for a new release even in case of corrective maintenance and configuration changes.

- **Portability** The current controller software supports VxWorks and Microsoft Windows NT. In the meantime, there is also a need of openness for choosing among different operating system (OS) vendors, e.g., Linux and Windows CE, and possibly new OS in the future.
- **Extensibility** The current controller software supports around 20 different applications that are developed by several distributed development centers around the world. To adapt to the increased customer focus on specific applications and to enable the establishment of new market segments, it was decided that the controller must constantly raise the service level through supporting more functionality and providing more features, while keeping important non-functional properties.
- **Testability** The controller software exposed huge number of public interfaces which resulted in tremendous time merely on interface tests. Therefore, it was decided to reduce the public interfaces to around 10% of the original value. Besides, due to the monolithic characteristic, error corrections in one part of the software required sometimes retesting of the whole system. One decision was therefore to investigate the feasibility of testing only modified parts.
- **Domain-specific attributes** The most important domain-specific attributes are related to real-time and potential problems with execution time. The critical real-time calculation demands from the controller software required reduced code size of the base software and runtime footprint.

### 5.1.3 Applying the Qualitative Evolvability Analysis Method

The main focus in our case study was to assess how well the architecture would support forthcoming requirements and understand their impact. The forthcoming requirements emerged due to the change stimuli from business strategy of the company:

- Time-to-market requirements, such as building new products for dedicated market within short time;
- Increased ease and flexibility of distributed development of diverse application variants.

Through the evolvability analysis process (as shown in Figure 4-3), we identified potential weakness in the original architecture, and defined an evolutionary path for the software system. The identification and analysis of the architectural requirements was performed by the architecture assessment core team which consisted of seven persons. It was a continuous maturation process from the first vision to concrete activities that will be described below. Three persons from the architecture core team identified the architectural solution proposals for some components in the *Base System* subsystem. I worked within the architecture assessment core team, and proposed potential architectural solutions that would facilitate the implementation of the identified architectural requirements. These proposals were discussed with the main technical responsible persons and architects. The choice of architectural solutions was based on discussions with the system architects and prototyping through the whole architecture assessment process. All the architectural decisions and solutions were documented and transferred further to the implementation teams.

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

Due to the aforementioned change stimuli, the main requirements on the software architecture and the refined activities for each requirement were proposed by the architecture core team, and are listed below:

- **R1**. Transform the monolithic architecture to modular architecture
  - Enable the separation of layers within the controller software: (i) a kernel which comprises of components that must be included by all application variants; (ii) common extensions which are available to and can be selected by all application variants; and (iii) application extensions which are only available to specific application variants.
  - Investigate dependencies between the existing extensions.
- **R2**. Reduced architecture complexity
  - Define system interfaces between subsystems and reduce the number of public interface calls.
  - Add support for real-time task isolation management.
  - Introduce a new scripting language to improve support for application development, since modern scripting languages are flexible, productive and reduce the need to recompile.
- **R3**. Enable distributed development of extensions with minimum dependency

  - Build the application-specific extensions on top of the base software (including kernel and common extensions) without the need of access and modification to the internal base source code.
  - Package the base software into Software Development Kit (SDK), which provides necessary interfaces, tools and documentation to support distributed application development.
- **R4**. Portability
  - Investigate portability across target operating system platforms.
  - Investigate portability across hardware platforms.
- **R5**. Impact on product development process
  - Investigate the implications of restructuring the automation controller software, with respect to product integration, verification and testing.
- **R6**. Minimized software code size and runtime footprint
  - Investigate enabling mechanisms, e.g., properly partitioning functionality.

These requirements were then checked against the evolvability subcharacteristics to justify whether the realization of each requirement would lead to an improvement of the subcharacteristics (or possibly a decrease, which would then require a tradeoff decision). Table 5-1 summarizes how the identified architectural requirements are related to the evolvability subcharacteristics.

**Table 5-1: Mapping between evolvability subcharacteristics and architecture requirements**

| Requirements | Subcharacteristics |
|---|---|
| R1. Transform the monolithic architecture to modular architecture.<br>R2. Reduced architecture complexity. | Analyzability |
| R1. Transform the monolithic architecture to modular architecture.<br>R2. Reduced architecture complexity. | Changeability |
| R3. Enable distributed development of extensions with minimum dependency. | Extensibility |
| R4. Portability. | Portability |
| R5. Impact on product development process. | Testability |
| R6. Minimized software code size and runtime footprint. | Domain-specific Attribute |

It may be noted that architectural integrity is omitted from this table. This is because, in the case study, architectural integrity was handled by documenting the architectural choices for handling potential architectural requirements, the rationales for the choice of architectural solutions along with their impacts on evolvability subcharacteristics. This will be detailed later.

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

With the consideration of not disrupting the ongoing development projects, the criteria for requirement prioritization were:

- Enable building existing types of extensions after refactoring and architecture restructuring;
- Enable new extensions, and simplify interfaces that are difficult to understand and/or may have negative effects on implementing new extensions.

Based on these criteria, R1, R2 and R3 were prioritized architectural requirements.

**Phase 2 - Step 2.1: Extract architectural constructs related to the respective identified issues**

We demonstrate the use of the analysis method by exemplifying with R3 (i.e., enable distributed development of extensions with minimum dependency). To enable distributed application development, there is a need to transform the existing system into components that can form the core of the product-line infrastructure, and separate application-specific extensions from the base software. Accordingly, we extracted architectural constructs that were related to the realization of distributed development. Details on how we go further with the extracted architectural constructs are described below in the following two steps.

**Phase 2 - Step 2.2: Identify refactoring components for each identified issue**

To enable distributed development of extensions with minimum dependency, the strategy of separate concerns was applied to isolate the effect of changes to parts of the system [12], i.e., separate the general system functions from the hardware, and separate application-specific functions from generic and basic functions. Based on the extracted cross-cutting concerns, the refactoring was conducted by merging subsystems/components, re-grouping of components, breaking down components and re-structuring them into new subsystems. Thus, the original architecture shown in Figure 5-1 was proposed to be changed to the architecture shown in Figure 5-2.

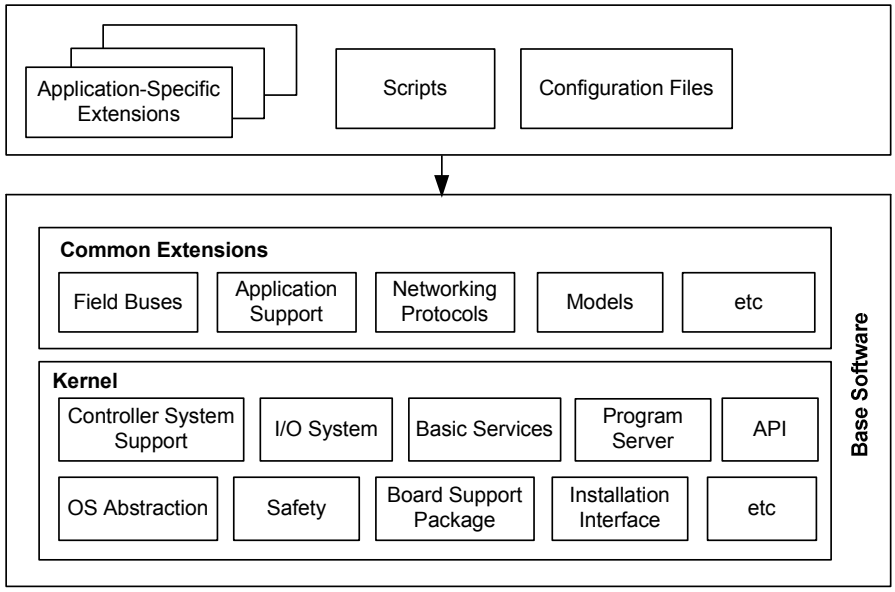


**Figure 5-2: A revised conceptual view of the software architecture**

Consequently, some subsystems and components need to be adapted and reorganized to enable the architecture restructuring. For instance, the *PC Applications* and *Man Machine Interaction* in the original architecture become *Application-Specific Extensions*, whereas the *OS & Hardware Abstraction* in the original architecture becomes a subsystem in the kernel in the new architecture. We also identified a collection of components that needed refactoring. Some of them were the components within the low-level basic services subsystem for resource allocations, e.g., *semaphore ID management* component, *memory allocation management* component. These components needed to be adapted because functionality needed to be separated from resource management in order to achieve the build- and development-independency between the kernel and extensions.

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

The complete assessment of components cannot be presented due to space limitations and company confidentiality. Therefore, we select a subset, and exemplify with one component that needed to be refactored. The example is chosen to be understandable for people outside the automation domain, while still representative and illustrative for the many various discussions and solutions that occurred during the analysis. We will focus on technical perspective, and discuss in terms of the following views:

- *Problem description*: the problem and disadvantages of the original design of the component;
- *Requirements*: the new requirements that the component needs to fulfill;
- *Improvement solution*: the architectural solution to design problems;
- *Architectural consequences*: the architectural implications of the deployment of the component on evolvability subcharacteristics.

**Component Example: Inter-Process Communication (IPC)**

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

- **Problem Description** All the slot names and slot IDs that are used by the kernel and extensions are defined in a C header file in the system. The developers have to edit this file to register their slot name and slot ID, and recompile. Afterwards, both the slot name and slot ID are specified in the startup command file for thread creation. There is no dynamic allocation of connection slot.
- **Requirements** The refactoring of this component is directly related to R3; It should be possible to define and use IPC slots in common extensions and application extensions without the need to edit the source code of the base software and recompile. The mechanism for using IPC from extensions must be available also in the kernel, to facilitate move of components from kernel to extensions in the future.
- **Improvement Solution** The slot ID for extension clients should not be booked in the header file. Extensions should not hook a static slot ID in the startup command file. The command attribute dynamic slot ID should be used instead. The IPC connection for extension clients will be established dynamically through the *ipc_connect* function as shown in Figure 5-3. It will return a connection slot ID when no predefined slot ID is given. An internal error will be logged at startup if a duplicate slot name is used.

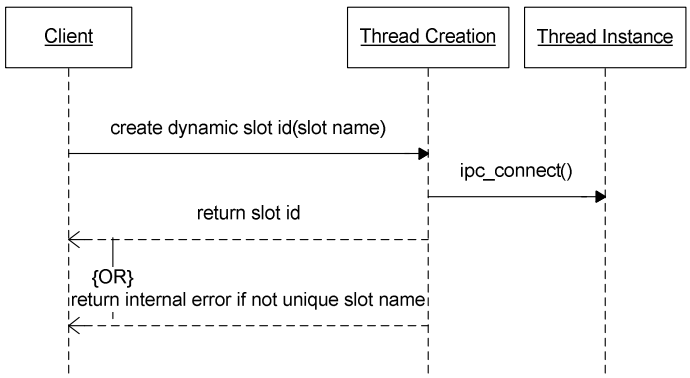


**Figure 5-3: Inter-process communication component after refactoring**

- **Architectural consequences** The revised IPC component provides efficient resource booking for inter-process communication, and enables encapsulation of IPC facilities. Accordingly, distributed development of extensions utilizing IPC functionality is enabled. The use of dynamic inter-process communication connections addressed resource limitations for IPC connection. In this way, limited IPC resources are used only when the processes are communicating. However, the use of IPC mechanisms dynamically requires resources, which are limited due to real-time requirements. This may require additional analysis including a trade-off analysis of possible solutions.

**Phase 2 - Step 2.4: Define test cases**

The corresponding test cases were derived based on the selected improvement solution to each component that needed refactoring. For instance, the architectural test cases for the IPC component are given by the *ThreadCreation* class creating dynamic slot ID, as shown in Figure 5-4.

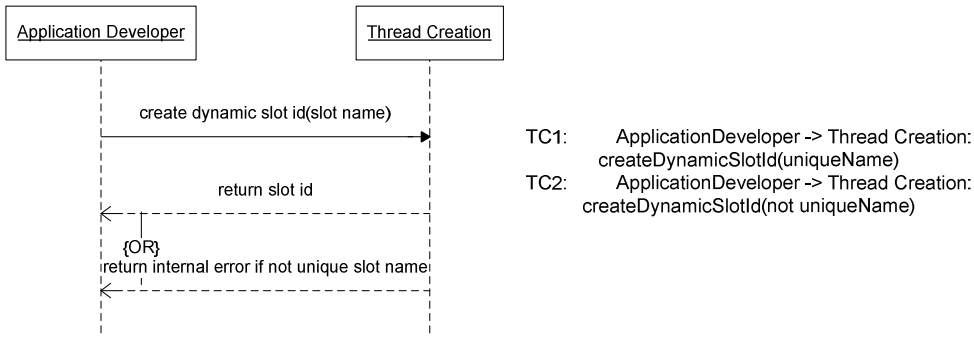


**Figure 5-4: Test cases for IPC component**

### Phase 3 - Step 3.1: Present evaluation results

Until this step, key architectural requirements were identified; components that needed to be refactored were identified; the stakeholders established a common understanding of potential improvement strategies and evolution path for the software architecture. In Table 5-2, we summarize the implications of the refactored IPC component on evolvability subcharacteristics.

**Table 5-2: Implications of the IPC component refactoring on evolvability subcharacteristics (+ positive impact, - negative impact)**

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

The qualitative analysis of the potential architectural solution's impacts on evolvability subcharacteristics provided the involved stakeholders with a good understanding of the corresponding tradeoffs when choosing architectural solution alternatives. Thus, an ad hoc choice of architectural solution can be avoided. As shown in Table 5-2, the negative impacts of the IPC component refactoring on evolvability subcharacteristics are not crucial.

### 5.1.4 Qualitative Evolvability Analysis: Experiences

Based on our experience in applying the qualitative evolvability analysis method, the architecture requirements, corresponding architectural decisions, rationale and potential architecture evolution path became more explicit, better founded and documented. Consequently, we have improved the capability of being able to understand and analyze systematically the impact of a change stimulus. This, in turn, helped us to prolong the evolution stage [21].

We list below two observations that concern visible improvements in the organization. They were also perceived and reported by the stakeholders themselves.

- **High-level business goals concretized into architectural requirements** High-level business goals lead to architectural requirements. In the case study, the potential requirements on the architecture were derived from the high-level business goals in the first phase of analysis, in which the potential architectural requirements were identified based on the change stimuli. Such derivation provided an understanding on how the intended software system and its evolving artifacts reflect and contribute to the strategic goals. Together with the documentation of architecture evolution path, it helped to enrich architectural models and facilitate the traceability of software architecture evolution back to the various business constraints and assumptions [15].
- **Improved documentation of architecture** The architecture transformation and suggestions for architectural solutions were part of the analysis process, which was performed by the architecture core team. Three persons from the architecture core team identified the architectural solution proposals for the components in the main subsystems over a six-month period. As a result of the analysis, the implementation solution proposals have been approved, and the documentation of architecture [91] [106] has been improved. The

final architectural analysis investigation report was distributed for inspection, and was approved after a few iterations. This document served as an input and blueprint to the implementation teams. In this way, the architecture core team and implementation teams shared the same view on the evolution path of the software architecture.

### 5.1.5 Qualitative Evolvability Analysis: Lessons Learned

In the qualitative evolvability analysis method, the architecture tradeoff analysis is reflected in two constituent steps:

- During architecture workshops, the stakeholders prioritize potential architectural requirements, which are mapped against evolvability subcharacteristics. By prioritizing the potential architectural requirements based on pre-defined criteria, evolvability subcharacteristics are implicitly prioritized by stakeholders;
- After the workshop, the identified architectural choices are qualitatively analyzed with respect to their impacts and support for evolvability subcharacteristics.

Therefore, we see two aspects which we can further explore and make more explicit:

- **Explicit stakeholders' views on prioritization and preferences on evolvability subcharacteristics**
  - Rationale:

  Depending on their roles that are involved in the development and evolution of a software system, the stakeholders usually have different concerns, i.e., interests which pertain to the system's development, its operation or evolution. Consequently, architecting for an evolvable software system implies that an architect needs to balance numerous stakeholders' concerns that are reflected in terms of their prioritization and preferences on evolvability subcharacteristics. When the prioritization and preferences of evolvability subcharacteristics are not explicitly expressed by involved stakeholders, it becomes difficult to determine the dimensions along which a system is expected to evolve.

  - Related activities performed in the qualitative evolvability analysis method:

  This aspect was treated implicitly in step 1.2 in the first phase (i.e., prioritize requirements on the software architecture), in which the

potential architectural requirements were mapped against evolvability subcharacteristics, and were then prioritized based on predefined criteria. As a result, the choice of prioritized architectural requirements implicitly sets a priority ranking on evolvability subcharacteristics.

- **Quantification of architectural solution alternatives' impacts on evolvability subcharacteristics**
  - Rationale:

  Choosing an architectural solution that satisfies evolvability requirements is vital to the evolution and success of a software system. Nonetheless, each solution candidate is associated with multiple attributes, as the choice of any solution alternatives may probably cause varied tradeoffs among evolvability subcharacteristics. Hence, it is important to understand how an architectural alternative supports different evolvability subcharacteristics, especially when there are several alternatives to choose among, each of which exhibits varied support for evolvability subcharacteristics. Consequently, these alternatives need to be ranked, and meanwhile, reflect stakeholders' preference information on evolvability subcharacteristics.

  - Related activities performed in the qualitative evolvability analysis method:

  The determination of potential architectural solutions, along with their impact on evolvability subcharacteristics was qualitatively handled in step 2.3 in the second phase (i.e., identify and assesses potential refactoring solutions) by examining the rationale of a solution proposal along with its architectural implications (positive or negative impact) of the deployment of the component on evolvability subcharacteristics.

Based on the above experiences, we further extended the qualitative evolvability analysis with quantification feasibility, and validated the quantitative evolvability analysis method in another industrial setting. This will be described in Chapter 5.2.

## 5.2 Case Study II. Quantitative Software Evolvability Analysis

This section describes the case study in which we applied the quantitative software evolvability analysis method. The system that we investigated is a mobile network node software architecture at Ericsson.

### 5.2.1 Context of the Case Study

The case study was based on an assessment of the mobile network architecture with respect to the evolvability of a logical node at Ericsson. The main purpose of the logical node is to handle control signaling for and keep track of user equipment such as mobiles using a certain type of radio access. This is a mature system that was introduced about ten years ago, and has been refined since then. The system is expected to be still on the market for years ahead, and thus, needs to be easy to maintain and evolve further on.

The system software[9] is divided into two levels:

- Platform level which consists of operating systems, a distributed processing environment and application support;
- Application level that comprises of a control system and a transmission system. The control system is designed to process high-level protocols and control user traffic data flow in the transmission system. The transmission system is responsible for transport, routing and processing of user traffic.

The case study focused on one of the challenges that the system needs to meet, i.e., In-Service Software Upgrade (ISSU). The system downtime is divided into planned and unplanned downtime. The planned downtime is imposed by maintenance routines, such as correction package loading. The unplanned downtime is imposed by automatic recovery mechanisms in the system and manual restarts of the system due to a system failure. The actual downtime for a network is largely dependent on the frequency of the planned downtime events. There are two scenarios connected with planned downtime:

[9] For reasons of confidentiality, no more details about the system are presented here.

- **Update of a release** The corrections to a release include either correction packages that are distributed to all customers, or single corrections that are made for specific customers only, and may be later included in the correction packages. These corrections are planned patches, and can be updated at runtime. During the update of a release, no configuration data needs to be changed or updated.
- **Upgrade of a release** A release is upgraded to a new release with changed characteristics of the network node, e.g., changes in node configuration parameters, major changes in software and hardware. This causes downtime of the node today. During an upgrade, when the new software has been installed, the node is restarted (automatically or manually), and the local configuration that a customer maintains is converted to a new format if needed.

A main driver of the design and evolution of the system is the achievement of non-stop operation with minimum service impact. Therefore the focus of our study is the second scenario which is the main cause of node downtime, because the node restart being part of each upgrade means service interruption for 5 to 10 minutes. The architecture must support this emerging requirement of In-Service Software Upgrade in order to evolve. The evolvability analysis in the case study focused on analyzing the impact on the current architecture and identifying its potential evolution path considering the emerging software upgrade requirement.

### 5.2.2 Evolvability Subcharacteristics from Case Perspective

We describe below each evolvability subcharacteristic in conjunction with the case study context:

- **Analyzability** The release frequency of the system in the case study is twice a year, with various new customer requirements, strategic functionality and characteristics implemented in each release. In addition, the software development organization is feature-oriented, i.e., the software developers are not grouped based on subsystems; instead, they are grouped to implement a certain feature, and therefore often need to work across various subsystems. This requires that the software system needs to be easily understood and have the capability to be analyzed in terms of the impact to the software by introducing a change.

  From ISSU perspective, it was decided that an ISSU solution should be easy to understand for the development organization.

- **Architectural Integrity** In the development of network and node system, several architectural design patterns, guidelines as well as design rules with respect to conformity, modeling and style guides have been articulated in architectural specification document. All these fundamental principles (including strategies and guidelines) govern the design and evolution of the system, and therefore are clearly defined and communicated. In addition to the strategies that guide software developers in order to fulfill requirements (i.e., features of direct value for a user), system strategies are also defined to fulfill non-functional attributes of high priority.

  From ISSU perspective, to enable ISSU implementation, it was decided that ISSU rules should be followed. It was also decided that whether a potential ISSU architectural design has any violations against these general design rules needs to be checked. If any ISSU component has to break the rules, it is essential to record the rationale for such design decision and strategy.
- **Changeability** From ISSU perspective, four aspects were concerned:
  - How well can other architectural changes fit into the ISSU solution;
  - Many kinds of application changes shall be possible without special upgrade code, e.g., backward compatible interfaces;
  - It shall be as easy as possible to write special upgrade code if needed;
  - How easy is it to change the ISSU solution itself once it is used.
- **Extensibility** The system must constantly raise the level of service by extending existing features or adding new ones. From ISSU perspective, one concern was to identify if there are any limitations when introducing the ISSU solution.
- **Portability** The current node software supports VxWorks and Linux on a number of hardware variants. In the future, a possible scenario could be to change operating system or support new hardware.
- **Testability** The system has a number of variants based on the selection of hardware configuration and capacity level of the node. Therefore, a main concern is the ease to test and debug parts of the system individually, and extract test data from the system. From ISSU perspective, three aspects were concerned:
  - Would ISSU influence the number of variants;

  - Would it be possible to conduct component tests;
  - Would it be possible to reproduce test cases.
- **Domain-specific Attributes** In this case study, two domain-specific attributes were identified:
  - *Capacity*, which is an attribute that describes the subscriber and throughput capacity with various radio access types. It depends on the traffic pattern and dimensioning of the operator network. A logical node is dimensioned for a certain load. The admission control functions and limits given by capacity licenses would limit the number of subscribers allowed to enter the node and the number of resources occupied by these subscribers. Besides, overload protection mechanisms are implemented in case of node internal failures, network failure, reconfiguration or wrong node dimensioning. From ISSU perspective, three aspects were concerned: (i) ISSU total time; (ii) capacity impact during ISSU; and (iii) capacity impact during normal execution.
  - *Availability*, which is an attribute that describes the ability to keep the node in service, i.e., to keep the downtime to a minimum. It is also called In-Service Performance (ISP) by the domain experts that we interviewed. The system needs to be tolerant against both hardware- and software-related failures so that the services provided by the node are always available. The recovery functions aim to provide a non-stop mode of operation of the system, i.e., to recover from both software and hardware failures with minimal inconvenience to the attached subscribers. From ISSU perspective, three aspects were concerned: (i) redundancy of critical components during ISSU; (ii) impact of ISSU solution's complexity; (iii) impact of software or hardware failures during upgrade.

### 5.2.3 Applying the Quantitative Evolvability Analysis Method

The main focus in our case study was to identify, with respect to the system function In-Service Software Upgrade (ISSU), which architecture alternative has the most potential for fulfilling the quality requirements of the system among a set of architecture candidates.

**Phase 1 – Step 1.1: Elicit stakeholders' views on evolvability subcharacteristics**

The change stimuli to the evolution of the node architecture in the case study came from the ever growing stringent requirement on In-Service-Performance. Based on the identified change stimuli, the high level architectural requirements were defined in order to evaluate potential architectural solution alternatives:

- The atomic component for which an upgrade is performed must have backward compatible interfaces during the upgrade;
- The old configuration data (including node-internal replicate data) format must be available during the whole upgrade;
- Replicated subscriber data format must be available on old format until the upgrade is finished;
- It must be known on which software version each atomic component executes;
- There must be a component which controls the upgrade and is aware of the progress.

Based on these high level architectural requirements from ISSU perspective, Table 5-3 summarizes how specific architectural requirements are related to the evolvability subcharacteristics.

**Table 5-3: Mapping between evolvability subcharacteristics and specific architecture requirements**

| Requirements | Subcharacteristics |
|---|---|
| R1. The ISSU solution should be easy to understand for the development organization. | Analyzability |
| R2. Many kinds of application changes shall be possible without special upgrade code, e.g., backward compatible interfaces. | Changeability |
| R3. Enable introduction of ISSU solution without limitations on extending existing features or adding new ones. | Extensibility |
| R4. Enable change of operating system or hardware. | Portability |
| R5. Enable the ease to test and debug parts of the system individually, and extract test data from the system. | Testability |
| R6. The ISSU total time, capacity impact during ISSU and normal execution should be within specified values. | Capacity |
| R7. Critical components need to have redundancy during ISSU.<br>R8. The impact of software or hardware failure during upgrade should be limited. | Availability |

It may be noted that architectural integrity is omitted from this table. This is because, in the case study, it was decided that architectural integrity would be handled by documenting the architectural choices for handling potential architectural requirements along with their impacts on evolvability subcharacteristics.

To elicit stakeholders' views on evolvability subcharacteristics, we performed interviews with key personnel and software designers to understand architectural challenges over the years in general, as well as the challenges that the architecture is facing due to various emerging requirements, e.g., distributed development, increased productivity by including more features in each product release. In addition, we interviewed various stakeholders to elicit their views on evolvability subcharacteristics, including three system architects, an operative product manager, the system owner, and two software designers involved in the logical node's development. These stakeholders possess a wide range of expertise, covering platform development, communication protocol, node configuration, monitoring and upgrade. The quantitative evolvability analysis method was presented to the stakeholders that were to be interviewed so that they would have a clear idea of the entire process, and understand the value of their contribution. The interviews were conducted separately for each stakeholder with the intension that his/her preference judgment should not be influenced by other people. The interviews were semi-structured, and the interviewees were free to discuss their main concerns about evolvability subcharacteristics from their perspective. We also extracted the stakeholders' views on important domain-specific attributes, i.e., capacity and availability.

**Phase 1 – Step 1.2: Extract stakeholders' prioritization and preferences of evolvability subcharacteristics**

We extracted the information on the stakeholders' preferences after we had gone through the list of evolvability subcharacteristics and clarified the definition of each subcharacteristic in their specific context. This was to ensure that each stakeholder's prioritization of evolvability subcharacteristics is made upon the same ground. We asked each stakeholder to provide us with the preferences of evolvability subcharacteristics from his/her own perspective. Table 5-4 shows a system architect's preferences on evolvability subcharacteristics.

**Table 5-4: Preferences on evolvability subcharacteristics provided by a software architect**

| | Analyzability | Integrity | Changeability | Extensibility | Portability | Testability | Availability | Capacity |
|---|---|---|---|---|---|---|---|---|
| **Analyzability** | 1 | 3 | 1/3 | 1/3 | 5 | 1 | 1/4 | 1/3 |
| **Integrity** | 1/3 | 1 | 1/6 | 1/6 | 2 | 1/3 | 1/8 | 1/7 |
| **Changeability** | 3 | 6 | 1 | 2 | 5 | 1/3 | 1/2 | 1/2 |
| **Extensibility** | 3 | 6 | 1/2 | 1 | 3 | 1/3 | 1/2 | 1/2 |
| **Portability** | 1/5 | 1/2 | 1/5 | 1/3 | 1 | 1/7 | 1/9 | 1/8 |
| **Testability** | 1 | 3 | 3 | 3 | 7 | 1 | 1/3 | 1/2 |
| **Availability** | 4 | 8 | 2 | 2 | 9 | 3 | 1 | 1 |
| **Capacity** | 3 | 7 | 2 | 2 | 8 | 2 | 1 | 1 |

After performing calculations based on equation (1) as described in Chapter 4.4.2, the values indicating subcharacteristic preference from the system architect's perspective are summarized in Table 5-5.

**Table 5-5: Subcharacteristics from an architect's perspective**

| Architect A | Analyzability | Integrity | Changeability | Extensibility | Portability | Testability | Availability | Capacity |
|---|---|---|---|---|---|---|---|---|
| **Preferences** | 0.077 | 0.030 | 0.135 | 0.110 | 0.023 | 0.158 | 0.249 | 0.219 |

These figures suggest that, from this system architect's perspective, the evolvability subcharacteristics are prioritized as (in declining order): availability (24.9%), capacity (21.9%), testability (15.8%), changeability (13.5%), extensibility (11%), analyzability (7.7%), integrity (3%), and portability (2.3%). Likewise, the other system architects' preferences on evolvability subcharacteristics were collected and calculated. Table 5-6 summarizes all the system architects' preferences on evolvability subcharacteristics, along with their aggregated prioritizations based on equation 2) as described in Chapter 4.4.2.

**Table 5-6: Aggregated subcharacteristics**

| Architects | Analyzability | Integrity | Changeability | Extensibility | Portability | Testability | Availability | Capacity |
|---|---|---|---|---|---|---|---|---|
| **Architect A** | 0.077 | 0.030 | 0.135 | 0.110 | 0.023 | 0.158 | 0.249 | 0.219 |
| **Architect B** | 0.059 | 0.047 | 0.084 | 0.064 | 0.057 | 0.105 | 0.407 | 0.176 |
| **Architect C** | 0.082 | 0.036 | 0.096 | 0.096 | 0.038 | 0.123 | 0.309 | 0.220 |
| **Aggregated** | 0.073 | 0.038 | 0.105 | 0.090 | 0.039 | 0.128 | 0.322 | 0.205 |

After the process of extracting system architects' preferences on evolvability subcharacteristics, it was interesting to note that the three system architects shared almost the same view of prioritization of evolvability subcharacteristics. They had commonly shared order of prioritization (starting from high to low priority) – availability (32.2%), capacity (20.5%), testability (12.8%), changeability (10.5%), extensibility (9%), and analyzability (7.3%). This is a good sign of the preference alignment among architects.

In the same way, we also gathered quality preferences for the other stakeholder roles, and realized that different stakeholder roles have different preferences of evolvability subcharacteristics. A summary of different stakeholder preferences is presented in Table 5-7.

**Table 5-7: Preferences on evolvability subcharacteristics provided by respective stakeholder roles**

| Stakeholders | Analyzability | Integrity | Changeability | Extensibility | Portability | Testability | Availability | Capacity |
|---|---|---|---|---|---|---|---|---|
| **Architects** | 0.073 | 0.038 | 0.105 | 0.090 | 0.039 | 0.128 | 0.322 | 0.205 |
| **Designers** | 0.105 | 0.125 | 0.103 | 0.108 | 0.042 | 0.154 | 0.322 | 0.041 |
| **System owner** | 0.061 | 0.189 | 0.111 | 0.108 | 0.023 | 0.112 | 0.350 | 0.046 |
| **Aggregated** | 0.080 | 0.117 | 0.106 | 0.102 | 0.035 | 0.131 | 0.331 | 0.098 |

The reason why we aggregate preferences per stakeholder role is that each role represents respective viewpoint and needs, and thus, we assume that the primary preference differentiation lies between the different stakeholder roles.

During the process in extracting stakeholders' views on evolvability subcharacteristics, we also performed consistency check for each stakeholder's comparisons based on AHP [156]. Table 5-8 summarizes the consistency ratio scores for each stakeholder.

**Table 5-8: Consistency ratios for stakeholders**

| Stakeholders | Architect A | Architect B | Architect C | Designers | System Owner |
|---|---|---|---|---|---|
| **Consistency Ratio** | 0.061 | 0.109 | 0.039 | 0.088 | 0.046 |

The research in [156] suggested that if the consistency ratio is smaller than 0.10, the participants' comparisons are consistent enough to be useful, and the AHP method can yield meaningful results. It is also pointed out in [156] that, in practice, higher values are often obtained, which indicates that 0.10 may be too hard. But it is an indication of the approximate value of the expected consistency ratio. As we see from Table 5-7, only architect B's value (0.109) is slightly more than 0.10. However, the value is still acceptable considering that 0.10 is a hard limit for the degree of consistency. Consequently, all the data we obtained from the stakeholders are trustworthy. The aggregated values of all the involved stakeholder roles, as shown in Table 5-5, indicate that availability has the highest priority, followed by testability, architectural integrity, changeability, extensibility, capacity, analyzability, and portability.

**Phase 2 – Step 2.1: Identify candidate architectural solutions**

Two architectural alternatives were developed for the In-Service Software Upgrade (ISSU) requirement in our study. For reasons of confidentiality we cannot give full descriptions of the candidate architectural solutions, but in principle, the architectural alternatives describe two variations of how to handle execution resource management. Two types of computing resources (processors) management are used to fulfill the capacity and In-Service Performance (ISP) requirements:

- Application processors that are optimized for node control and traffic control logic;

- Device processors that are optimized for communication/protocol logic to handle time-critical traffic data flow and control signaling termination.

Specifically, the two candidate architectural solutions are:

- **Alt1**: Slot by slot concept

  The overall idea is to take one board after another out of service for upgrade to a new release. During In-Service Software Upgrade, the boards running with old software will coexist and interact with the boards running with new software.

- **Alt2**: Zone concept

  The overall idea is to divide the node into two zones, i.e., in one zone, all components run old software, and in the other zone, components run new software.

Both solutions have their respective benefits and drawbacks. For instance, the slot by slot concept has the benefit of having board redundancy under control during ISSU and that the existing mechanisms in the architecture facilitate the potential implementations of ISSU. On the other hand, the zone concept has the benefit that backward compatibility is not needed for application and device processors. But both solutions face several drawbacks such as time required for ISSU, interface changes, and other specific ones. Therefore, it was not an easy task to directly decide which alternative would be more optimal than the other.

**Phase 2 – Step 2.2: Assess candidate architectural solutions' impacts on evolvability subcharacteristics**

To assess ISSU candidate architectural solutions' impacts on evolvability subcharacteristics, we actively cooperated with the system architects at Ericsson. The two candidate architectural solutions were rated with respect to how well they support each evolvability subcharacteristic. This information was provided by the three system architects because they possess the whole system perspective and technical knowledge. The values indicating the support weights of the two alternatives with respect to evolvability subcharacteristics are summarized in Table 5-9.

**Table 5-9: Support weights of the two alternatives on evolvability subcharacteristics**

| | Alt 1: Slot by slot concept | Alt 2: Zone concept |
|---|---|---|
| **Analyzability** | 0.667 | 0.333 |
| **Integrity** | 0.500 | 0.500 |
| **Changeability** | 0.250 | 0.750 |
| **Extensibility** | 0.333 | 0.667 |
| **Portability** | 0.500 | 0.500 |
| **Testability** | 0.333 | 0.667 |
| **Availability** | 0.750 | 0.250 |
| **Capacity** | 0.800 | 0.200 |

**Phase 3 – Step 3.1: Present evaluation results**

Until this step, key domain-specific attributes and candidate architectural solutions were identified; stakeholders' preferences on evolvability subcharacteristics as well as each candidate solution's support on evolvability subcharacteristics were quantified. Consequently, considering the prioritization weights of evolvability subcharacteristics in Table 5-6, together with the values indicating each alternative's support on evolvability subcharacteristics shown in Table 5-8, the overall weight for Alt1 is calculated based on equation (3) as:

$W_{Alt1}$ = 0.080 ×0.667 + 0.117 ×0.500 + 0.106 ×0.250 + 0.102 ×0.333 + 0.035 ×0.500 + 0.131 ×0.333 + 0.331 ×0.750 + 0.098 ×0.800 = 0.560

Likewise, $W_{Alt2}$ = 0.440, which indicates that, Alt1 (slot by slot concept) seems to be the preferred solution supporting evolvability.

## 5.2.4 Quantitative Evolvability Analysis: Experiences

By applying the quantitative evolvability analysis method, we have improved the capability of being able to explicitly extract stakeholders' views on evolvability subcharacteristics and quantify candidate architectural

solutions' support on evolvability. In this way, intuitive choices of architectural solutions are avoided during software evolution.

We list below two visible benefits that were perceived and reported by the involved stakeholders in the organization:

- **Quantification of stakeholders' preferences on evolvability subcharacteristics**

In this case study, different stakeholder roles had different concerns on the software system. For instance, the software designers mentioned three main aspects that were considered important from their perspective, i.e., functionality, ease to understand, and source code level performance; whereas the operative product manager focused on domain-specific attributes (i.e., availability and capacity), functionality, and time-to-market/time-to-customer. These concerns are aspects that are critical from specific stakeholder's perspective, and thus, will influence how he/she would prioritize evolvability subcharacteristics. According to the stakeholders that we interviewed, to think in terms of "subcharacteristics", was not new for them. But previously they had not been able to quantify the importance of the various -abilities for their system. The quantitative evolvability analysis method provided a structured way to extract and quantify the opinions of the stakeholders of various roles involved in the software architecture decision process through individual discussions and interviews. In addition, the quantification results served also as a communication vehicle for discussions of development concerns among various stakeholders when individual preferences were quantitatively identified and highlighted.

- **Quantification of architectural alternatives' impacts on evolvability**

In this case study, recalling the stakeholders' preference weight of evolvability subcharacteristics and the weight of how well different alternatives support a specific evolvability subcharacteristic, we obtained a normalized value, designating the overall weight for each alternative's support on evolvability, and indicating which was the preferred candidate architectural solution. In addition, we also interviewed the system architects after the execution of the method in the form of a discussion meeting to collect their opinions on if the method had produced relevant results. According to them, these results can definitely serve as a basis for further discussions on the choice of architectural solution. Most importantly, the systematic character of the evolvability analysis approach, including documentation of the reasoning of prioritization in each step, was most valuable, as it provided them an active countermeasure against arbitrarily

making some design decisions that would be otherwise often based on intuition because of personal experience and available expertise.

### 5.2.5 Quantitative Evolvability Analysis: Lessons Learned

In the case study, we conducted a series of semi-structured interviews with the stakeholders that participated in the evolvability analysis. During the interviews, we asked questions that were meant to extract and clarify the stakeholders' perception on evolvability subcharacteristics. In this process, cost was not explicitly considered. Cost involves development cost, maintenance and evolution cost, and concerns time-to-market. We put cost into consideration when candidate architectural solutions had been identified as it became more concrete to estimate the workload for each solution. On the other hand, in order to carry out software evolution efficiently, the cost aspect could also have been considered upfront and explicitly evaluated together with the evolvability subcharacteristics.

## 5.3 Summary

This chapter has described the application of software evolvability model, qualitative and quantitative architecture evolvability analysis methods in two industrial projects driven by the need of improving software evolvability. Based on our experiences, both the qualitative and quantitative analysis methods can be used as an integral part of software development and evolution process.

Throughout the process of evolvability analysis at ABB, the architecture requirements and corresponding design decisions for the transition of architecture became more explicit, better founded and documented. The resulting analysis results were well accepted by the stakeholders involved in the analysis process, and became a blueprint for further implementation improvement.

Throughout the process of evolvability analysis at Ericsson, the importance of various quality attributes perceived among different stakeholders was quantified and became more explicit. This quantification also served as a communication vehicle for further discussions among stakeholders.

In both cases, by analyzing architectural improvement proposals with respect to their implications on evolvability subcharacteristics, we further avoided an ad hoc choice of potential evolution paths of software architecture.

Another remark is that we plan to further complement the quantitative analysis method with cost aspect more explicitly to better support design decisions, and validate on additional, independent cases.

# Chapter 6. Open Source Software Evolution

Up to this point, we have laid the foundations for analyzing software architecture evolvability, principally the broad set of studies in architecting for evolvability during software architecture evolution (Chapter 3), the proposed software evolvability model and evolvability analysis process that comprises both qualitative and quantitative analysis methods (Chapter 4). Chapter 5 presented case studies to cement the proposed evolvability model and evolvability analysis process. As stated in Chapter 1, the focus of our research is to analyze proprietary systems for software evolvability improvement. However, as a supplemental research contribution, we now turn our attention to the open source software evolution.

With the emergence of the Open Source Software (OSS) paradigm, researchers have access to the code bases of a large number of evolving software systems along with their release histories and change logs. There have been a large number of studies published on OSS characteristics and evolution patterns by examining sequences of code versions or releases using statistical analysis. Meanwhile, the easily accessible data about different aspects of OSS projects also provides researchers with immense number of opportunities to validate the prior studies of proprietary software evolution [112] and to study how evolvability has been addressed in OSS evolution.

This chapter presents the results from a systematic literature review (SLR) that we performed in the area of OSS evolution research, and will cover three aspects:

- Systematically select and review published literature in order to build and present a holistic overview of the existing studies on OSS evolution.
- Analyze the literature to find out how software evolvability is addressed during development and evolution of OSS.

- Extract information on the metrics that researchers use for measuring OSS evolution from different perspectives such as growth patterns, complexity patterns, processes and evolution effort estimation.

The detailed research questions include:

- What are the main research themes that are covered in the scientific literature regarding open source software evolution, and analysis and achievement of evolvability-related quality attributes?
- What are the metrics that are used for OSS evolution measurement and analysis, and what are the limitations in using these metrics, if any?

The rest of this chapter is structured as follows: Chapter 6.1describes the research method used. Chapter 6.2 presents demographic information of the primary studies included in the review. Chapters 6.3 to 6.6 discuss the findings from this systematic review. Chapter 6.7 discusses validity threats of the review.

## 6.1 Systematic Literature Review Process

This research was undertaken as a systematic review [101], which is a process of assessing and interpreting all available research related to a particular research topic. The process consists of several stages:

- Development of a review protocol;
- Identification of inclusion and exclusion criteria;
- Searching relevant papers;
- Data extraction and synthesis.

These stages are detailed in the following subsections.

### 6.1.1 Review Protocol

The review protocol was designed based on the Systematic Literature Review (SLR) guidelines by Kitchenham and Charters [101]. The protocol specified the background for the review, research questions, search strategy, study selection criteria, data extraction and synthesis of the extracted data. The protocol was developed mainly by one researcher, and reviewed by me and another senior researcher to reduce bias.

### 6.1.2 Inclusion and Exclusion Criteria

The inclusion and exclusion criteria mainly focused on including full-text papers in English from peer-reviewed journals, conferences, workshops and book chapters published until the end of 2009. We exclude studies that do not cover evolution of open source software, prefaces, and articles in the controversial corner of journals, editorials, and summaries of tutorials, panels and poster sessions.

### 6.1.3 Search Process

The search strategy was designed to search in a selected set of electronic databases:

- ACM Digital Library (http://portal.arm.org)
- Compendex (http://www.engineeringvillage.com)
- IEEE Xplore (http://www.ieee.org/web/publications/xplore/)
- ScienceDirect – Elsevier (http://www.elsevier.com)
- SpringerLink (http://www.springerlink.com)
- Wiley InterScience (http://www3.interscience.wiley.com)
- ISI Web of Science (http://www.isiknowledge.com)

The search terms used for constructing search strings were: "open source software" OR "libre software" OR "free software" OR "FOSS" OR "F/OSS" OR "F/OSSD" OR "FOSSD" OR "FLOSS" OR "F/LOSS" OR "OSSD".

The selection of studies was performed through a multi-step process:

- Searches in the databases to identify relevant studies by using the search terms;
- Exclude studies based on the exclusion criteria;
- Exclude irrelevant studies based on titles and abstracts;
- Obtain primary studies based on full-text reading.

The searches in electronic databases were performed in two stages. At the first stage, the papers published until the end of 2008 were searched, and then a separate complimentary search was performed for 2009 publications. After merging the search results and removing duplicates, there were 11,439 papers published until 2008 and 1,921 papers published in 2009. After scanning all the papers by titles and abstracts, 134 papers were selected. In the final stage, full-text was scanned, and we selected 41 papers for this

review. The paper selection process involved two researchers (me and another researcher) to decide whether to include or exclude a paper. A paper was excluded if both researchers considered it irrelevant. Any disagreement was resolved through discussions and involvement of a third senior researcher.

### 6.1.4 Data Extraction and Synthesis

Data extraction and synthesis were carried out by reading each of the 41 primary studies thoroughly and extracting relevant data, which were managed through bibliographical management tool EndNote and Spreadsheets. The data extraction was driven by a form show in Table 6-1. For data synthesis, we inspected the extracted data for similarities in terms of the focus of the studies in order to define how results could be compared. The results of the synthesis will be described in the subsequent chapters.

**Table 6-1: Data extraction for each study**

| Extracted Data | Description |
|---|---|
| Study identity | Unique identity for the study |
| Bibliographic references | Author, year of publication, title and source of publication |
| Type of study | Book, journal paper, conference paper, workshop paper |
| Focus of the study | Main topic area and aspect of open source software being investigated |
| Research method used for data collection | Included technique for the design of the study, e.g. case study, survey, experiment, interview to obtain data, observation |
| Data analysis | Qualitative or quantitative analysis of data |
| Metrics used | The metrics used in data collection for analysis |
| Constraints and limitations | Identified constraints and limitations in each study |

## 6.2 Overview of the Primary Studies

This chapter provides some demographic information about the primary studies. Chapter 6.3 to 6.6 will present the findings from analyzing the data extracted from the reviewed studies in order to answer the research questions which motivated this systematic literature review.

### 6.2.1 Demographic Information of the Primary Studies

It has been mentioned that we performed searches in multiple electronic databases. We found that the largest numbers of selected papers (22 papers) were published on OSS evolution from IEEE. The second largest numbers of papers (9 papers) were published by ACM; while four selected papers were published by John Wiley & Sons in its Journal of Software Maintenance and Evolution. Trend of publications over the years shows a positive growth except for year 2008. Only three papers on OSS evolution were published in that year. In year 2009, eleven papers were published showing that a good number of researchers are addressing OSS evolution.

Our review has also found that the evolution trends and patterns is the most focused research area with 23 papers published on this topic. There were 10 papers on the role of process support in evolution. However, few papers address the characteristics of evolvability and architecture, with 5 and 3 papers respectively.

### 6.2.2 Categories of the Primary Studies

As described in Chapter 6.1.4, during the data synthesis phase, we examined the papers based on their similarities in terms of research topics and contents in order to categorize the primary studies of OSS evolution. Besides classifying the primary studies, we examined also the metrics used for assessing OSS evolution as well as the analysis methodology for collected data in each study.

After examining the research topics, data analysis and findings addressed in each study, we identified four main categories of themes, one of which is further refined into sub-categories to group primary studies that share similar characteristics in terms of specific research focus, research concepts and contexts. The categories and sub-categories are:

- OSS evolution trends and patterns
    - Software growth
    - Software maintenance and evolution economics
    - Prediction of software evolution
- OSS evolution process support
- Evolvability characteristics
- Examining OSS evolution at software architecture level

These themes and their corresponding sub-categories will be further detailed in the following chapters. For each category of theme, we will describe the category and related studies along with the metrics that are used to quantitatively or qualitatively analyze the OSS evolution. Finally an analysis of the studies is discussed with main findings summarized.

## 6.3 OSS Evolution Trends and Patterns

This category includes studies that focus on investigating OSS evolution trends and patterns. Based on their focus, the studies were further classified into three sub-categories:

- Software growth
- Software maintenance and evolution economics
- Prediction of software evolution

### 6.3.1 Software Growth

The studies in this sub-category mainly focus on software growth and changes using a variety of metrics as shown in Table 6-2.

**Table 6-2: Software growth metrics**

| Study | Metrics |
|---|---|
| [2] | Number of packages, number of classes, total lines of code, number of statements |
| [4] | Types of extracted changes: addition of source code modules in successive versions of software; deletion; and modification |
| [40] | Initial size, current size, modules (folders), modules (files), average module size, days through versions, versions, version rate, delta size |
| [46] | Source file, source folder, source tree, size, RSN (release sequence number), level number, depth of a folder tree, width of a level, width of a folder tree, files added, modified or deleted |
| [81] [149] | Lines of code (LOC) in source files as a function of the time in days |
| [82] | Lines of source code, the number of packages, the changed and unchanged packages |
| [90] | LOC (lines of code), number of directories, total size in Kbytes, average and median LOC for header and source files, number of modules (files) for each subsystem and for the system as a whole |
| [102] | Number of LOC added to a file, including all types of LOC, e.g. also commentaries |
| [138] | Overall project growth in functions over time, overall project growth in LOC over time |
| [152] | Lines of source code, the number and size of packages |
| [162] | Lines of code (LOC), executable LOC, lines of code per comment ratio, functions added over each release, number of functions |
| [164] | Size in number of source code files, number of files handled (added, modified, deleted) between two subsequent releases, average complexity |
| [169] | Rate of growth with respect to release sequence number |
| [176] | Module, bugs, bug fixing and requirement implementation |
| [180] | Source code metrics, e.g., lines of code, number of modules, number of definitions |

According to Koch [102], software growth modeling can be of interest for developing models to predict software evolution, maintainability and other characteristics. Moreover, many OSS studies focus on utilizing the OSS evolution data to verify Lehman's laws of software evolution [105]; their findings either conform or diverge from the growth behavior of proprietary software. It is essential that the measures of software growth can actually represent and quantify the notion of software growth in order to obtain a reasonable comparison among the results from different studies. However, we noticed that there have been conflicting interpretations of some important operational definitions with respect to the metrics used for measuring software growth patterns. Some examples of the operational definitions that

exhibit varying interpretations include system growth, system change, and size, as discussed below:

- **System growth**

Software growth is measured by using the metric of percentage growth over time. There exist diverse interpretations of rate of growth. For instance, one assumption in some empirical studies [159, 162] on software evolution, as also suggested by Lehman [112], is to analyze and plot growth data with respect to the release sequence number (RSN).

Another interpretation of rate of growth is reflected in the study by Godfrey and Tu [81], who plotted growth rates against calendar dates rather than release numbers. Furthermore, they suggest that plotting against release numbers would have led to dips in the function curves because development and stable releases follow different behaviors. This interpretation of rate of growth is further confirmed by Thomas et al. [169], who came to the conclusion that due to the new temporal variables introduced by OSS, the rate of growth of OSS should be computed with respect to temporal variables such as the release date. It was also validated that different conclusions can be drawn when software evolution data are analyzed against release date rather than RSN. Therefore, diverse interpretations of rate of growth can pose a threat in properly interpreting the OSS evolutionary behaviors.

- **System change**

Separating the characterizations of system growth and system change is a challenge [112]. A variety of change metrics can be used. For example, Xie et al. [180] used changes to program elements (such as types, global variables, function signatures, and bodies) to characterize system change. Cumulative numbers of addition and deletion types of changes to these program elements are plotted. They reported that the majority of changes are made to functions.

It is also possible to count all the different files that have been added, modified and deleted between two subsequent releases in order to measure system changes [164]. In this case, the conventions used for measuring changes can lead to different results in interpreting the OSS evolutionary behaviors, e.g., whether or not taking into consideration of the changes in comment lines or minor changes in a single source line.

- **Size**

Lehman suggests using the number of modules to quantify program size, as he argues that this metric is more consistent than considering source lines of

code [112]. However, there are different interpretations of a module. For instance, Simmons et al. [162] consider modules only at the file level; while Capiluppi [40] studies both at file level and directory level, and discovers different OSS evolutionary behaviors depending on whether directories or files are considered as modules.

Instead of using modules as Lehman suggested, LOC (lines of code) is often used for measuring the size of OSS. For instance, Godfrey and Tu [81] used number of uncommented lines of code because as they claim, using number of source files would have meant losing some of the full story of the evolution of the system, especially at the subsystem level due to the variation in file sizes. Conly and Sproull [81] also assume that the total number of uncommented LOC grows roughly at the same rate as the number of source files. However, this assumption is not fully validated in a broader scope as it was only verified in some of the largest packages in Debian GNU/Linux [86].

Moreover, the definition of LOC varies as different studies interpret LOC differently, depending on the tools and available data sources used [136]. Koch's definition of LOC [102] considers all types of files, including comments and documentation. Some other studies [81, 149] count LOC in two ways: including blank lines and comments in source files (e.g., in .c and .h files), or ignoring blank lines and comments. This kind of counting applies only to source files, and ignores other source artifacts such as configuration files, make-files, and documentation.

Even the term *source file* is defined in different ways. For example, Smith et al. [164] consider only files with extension .c as source files. Therefore, for systems involving a variety of source file extensions, different assumptions regarding file extensions and their belonging to the source code or not could lead to different values in size, which would affect the analysis results of different aspects of evolutionary behaviors [151].

### 6.3.2 Software Maintenance and Evolution Economics

The uncertainties in software evolution arise from, to a certain extent, understanding how OSS would have evolved in terms of costs. Moreover, software evolvability concerns both business and technical perspectives, as the choice of maintenance decisions from technical perspective needs to be balanced with economic valuation to mitigate risks. Therefore, another perspective in understanding OSS evolution trends is to analyze how software has evolved in terms of development and maintenance costs.

Capra et al. [48] analyze the quality degradation effect, i.e., entropy of OSS by measuring the evolution of maintenance costs over time. The metric used in this study is function points, and is based on the assumption that the maintenance costs are proportional to the time elapsed between the releases of two subsequent versions. Another study by Capra [47] proposes an empirical model to measure evolutionary reuse and development cost which is an indicator of the effect of maintenance decisions made by OSS developers. The metric used is source lines of code (SLOC).

### 6.3.3 Prediction of Software Evolution

The OSS history data over time can be utilized to predict its evolution. It has been mentioned in Chapter 6.3.1 that modeling software growth is essential for developing software evolution prediction models. Although there are many studies of monitoring OSS growth, comparatively fewer studies actually utilize the historical evolution data for the purpose of predicting its evolution.

We find only three papers in this area. Herraiz et al. [85] describe using data from source code management repository to compute size of the software over time. This information is used to estimate future evolution of the project. SLOC is used for counting program text that is not a comment or blank line regardless of the number of statements or fragments of statements on the line. All lines that contain program headers, declarations, and executable and non-executable statements are excluded. Therefore, the results may vary if other sorts of files are considered.

Yu [184] uses source code changes to indirectly predict the maintenance effort of OSS. The metrics used include lag time between starting a maintenance task and closing the task, source code change at module level (e.g., number of modules added, deleted and modified), and source code change at line level (e.g., number of source LOC added, deleted and modified) in one maintenance task. Some threats in this study are that all module-level changes are treated in the same manner irrespective of the amount of changes as well as the effort for line-level changes.

Another way to predict OSS evolution was proposed by Raja et al. [143], who described using data from monthly defect reports to build up time series model that can be used to predict the pattern of OSS evolution defects.

## 6.4 Evolution Process Support

This category includes studies that focus on OSS evolution support from various perspectives of software development process.

- Feedback-driven quality assessment

Bouktif et al. [29] propose an approach that is based on remote and continuous analysis of OSS evolution. This approach utilizes available data sources such as CVS versioning system repository, commitment log files and exchanged mails in order to provide services that mitigate software degradation and risks. The principle services include growth, complexity and quality control mechanism, feedback-driven communication service, and OSS evolution dashboard service.

- Commenting practice

To understand the processes and practices of open source software development, Arafat and Riehle [7] treat the amount of comments in a given source code body as an indicator of its maintainability. They focus on one particular code metric, i.e., the comment density. According to them, *commenting* practice is an integrated activity in OSS development, and successful OSS projects follow consistently this practice.

- Exogenous factors

Capiluppi and Beecher [41] investigated whether or not an OSS system's structural decay can be influenced by the repository in which it is retained. Based on a comparative analysis of two repositories, they concluded that the repositories in which OSS are retained act as exogenous factors, which can be a differentiating factor in OSS evolvability. Beecher et al. [20] extended that work by involving more repositories and strengthening the results with the formulation of different types of OSS repository along with a transition framework among the various types.

Robles et al. [150] describe the problems that can be found when retrieving and preparing for OSS data analysis, and present the tools that support data retrieval for OSS evolution analysis such as source code, source code management systems, mailing lists, and bug tracking systems. In accordance with this study, Bachmann and Bernstein [13] address the quality of data sources and provide insights into the influencing factors to the quality and characteristics of software process data gathered from bug tracking database and version control system log files. These studies reflect that the analysis of the evolution and history of an open source software as well as the prediction of its future rely on the quality of data sources and corresponding process data.

- Maintenance process evaluation

Koponen [104] presents an evaluation framework for OSS maintenance process. The framework includes attributes for evaluating activity, efficiency and traceability of defect management and maintenance processes.

- Evolution model

The traditional staged model [21] represents the software lifecycle as a sequence of stages. Instead of using the model that was built mainly by observing traditional software development, Capiluppi et al. [43] revised the staged model for its applicability to OSS evolution.

- Configuration management

Asklund and Bendix [11] examine the configuration management process, and analyze how process, tool support, and people aspects of configuration management influence the OSS evolution.

## 6.5 Evolvability Characteristics

This category includes studies that focus on characteristics that can be considered important for software evolvability.

### 6.5.1 Determinism

As indicated by Herraiz et al. [84], the evolution of open source projects is governed by a sort of determinism, i.e., the current state of the project is determined time ago. Their results also show that at least 80% of the sampled projects are short-term correlated. However, a long-term perspective to explicitly address evolvability for the entire software lifecycle is required since the inability to effectively and reliably evolve software systems means loss of business opportunities [21].

### 6.5.2 Code Understandability

Another OSS evolvability characteristic is code understandability [44]. This study views understandability as a key aspect for maintainability, and takes into account only code structure measures (e.g., code size, number of macro-modules and micro-modules, size of modules, and average size of modules) for calculating code indistinctness as an indicator of code understandability.

### 6.5.3 Complexity

Complexity is a software characteristic that affects evolvability. Table 6-3 summarizes the variety of metrics that have been used to characterize OSS evolution from software complexity perspective.

**Table 6-3: Complexity metrics**

| Study | Metrics |
|---|---|
| [2, 42, 58] | McCabe's cyclomatic complexity |
| [45] | System size and the evolving structure of the software |
| [46] | McCabe's cyclomatic complexity for structural complexity, Halstead Volume for textual complexity |
| [138] | Overall project complexity, average complexity of all functions, average complexity of functions added |
| [162] | Overall release complexity and average function complexity using McCabe and Halstead complexity measure |
| [180] | McCabe's cyclomatic complexity, common coupling and average number of function calls per function |

According to Table 6-3, *McCabe's cyclomatic complexity* [124] is the most often used metric. It measures the number of independent paths in the control flow graph. The rationale for using this metric is that the number of control flow paths is correlated to how well-structured the functions are in the program. Another metric is *Halstead complexity*, which measures a program module's complexity directly from source code, with focus on computational complexity. These two complexity measures have different emphasis, and therefore, can be complementarily used. For instance, Simmons et al. [162] found that the *McCabe* and *Halstead complexity* metrics yielded contradictory results, which suggested that while the structure complexity declines with successive releases, the complexity of calculation logic increases.

Besides *McCabe* and *Halstead* indexes, there are other additional indicators of complexity, both at system and component level, as well as function level:

- *Calls per function*, indicating the complexity of functions. It is computed by averaging the number of calls per function for all functions [180].
- *Coupling*, representing the number of inter-module references.

- *Interface complexity*, measuring the sum of input arguments to, and return states from, a function [167]. The number of arguments and state returns has impact on software changeability.
- *Complexity* of some systems may also be found in their *data structures* rather than in source code [138].

However, we did not find any research papers that explicitly study complexity in terms of coupling, interface complexity and data structure complexity.

### 6.5.4 Modularity

Modularity is a concept by which a piece of software is grouped into a number of distinct and logically cohesive sub-units, presenting services to the outside world through a well-defined interface [16]. Table 6-4 summarizes the metrics that have been used to characterize OSS evolution from modularity perspective. It is obvious from this table that the metrics for modularity are used at different levels. For instance, Liu and Iyer [115] and Simmons et al. [162] studied modularity at the class/file level that provides information regarding software functionality. However, Conley and Sproull [58] argue that studying modularity at that level does not capture interface information, i.e., whether classes or files communicate via interfaces, which are used to achieve component independence in modular software. Accordingly, they argue that the package at the module or component level is more appropriate for assessment of software modularity than using classes or files.

**Table 6-4: Modularity metrics**

| Study | Metrics |
|---|---|
| [58] | Total number of lines of code, number of concrete and abstract classes, afferent and efferent coupling |
| [82] | Dependencies between packages |
| [115] | Measured at class/file level |
| [138] | Correlation between functions added and functions modified |
| [162] | (Only measured at file level): number of classes, number of files for each release, directory structure and content |

Excessive inter-module dependencies have long been recognized as an indicator of poor software design [37], and can diminish the ability to reason about software components in isolation. It becomes also difficult to assess

and manage change impacts. Therefore, apart from studying the dependencies between packages [82], inter-module dependency can also be used for achieving modularity, and for examining the following kinds of dependencies:

- *Class reference*: If class A refers to class B, e.g., as in an argument in a method, then A depends on B.
- *Invokes*: If a function in class A calls a function or a constructor of class B, then A depends on B.
- *Inherits*: If class A is a subclass of class B, then A depends on B.
- *Data member reference*: If a function in class A makes reference to a data member of class B, then A depends on B.

However, we did not find any paper that explicitly studies OSS evolution by using the inter-module dependency.

## 6.6 Examining OSS at Software Architecture Level

This category includes studies that focus on examining OSS evolution at software architecture level. According to Nakagawa et al. [129], there is a lack of research that investigates the relation between software architecture and OSS, and discusses in details how software architecture is treated in OSS. Godfrey and Tu [81] came up with similar observations from another perspective, i.e., planned evolution and preventive maintenance may suffer in OSS development, which encourages active participation but not necessarily careful reflection and reorganization. The scarcity of studies on architectural-level evolution of OSS confirms the above-mentioned observations.

Based on a case study, Nakagawa et al. [129] found that software architecture is directly related to OSS quality, and that the knowledge and experience in architecture must be considered in OSS projects. This study also proposes architecture refactoring in order to repair architectures, and aims at improving mainly maintainability, functionality and usability of OSS. Tran et al. [171] describe a similar approach, and explains the process of forward and reverse architectural repair to avoid architectural drift.

There are not many measures proposed for the architectural level evolution. Some variants of the number of calls into and number of calls from a component are used in [41], which addresses the structural characteristics of

OSS with respect to the organization of the software's constituent components. This study selects functions as the basic unit for analysis, and three attributes are considered as proxies of static architectural structure, i.e., fan-in, fan-out and instability.

## 6.7 Summary

This chapter has presented the results from our systematic review which was based on 41 identified primary studies in open source software evolution. Based on the research topics of these studies, we have classified them into four main categories of themes:

- **Software trends and patterns**

Most papers focus on using different metrics to analyze OSS evolution over time. Few papers have looked into the economic perspective, e.g., maintenance effort, and few papers utilize the historical evolution data for prediction of OSS evolution and development. In this category, researchers have used various metrics at varying levels of granularities, e.g., class level, file level, and module level to measure OSS evolution. However, this review has also shown that there are diverse interpretations of the same terms, e.g., module, lines of code, rate of growth. This may cause conflicting conclusions that may be drawn from OSS evolution patterns, especially if the studies attempt to make comparisons on the differentiating results though based on using different sets of metrics for measuring.

- **Evolution process support**

Different aspects that appear to have impact on the OSS evolution process are covered, including commenting practice, OSS evolution and maintenance evaluation model, structures and quality characteristics of resources such as repositories, mails, bug tracking systems, as well as tools that support data retrieval for evolution analysis.

- **Evolvability characteristics**

Determinism, understandability, modularity and complexity are addressed in the included studies. However, there are more evolvability characteristics that are not covered such as changeability, extensibility, and testability. This might also explain the findings in the analysis of OSS evolution trends category that focuses on the evolution history instead of predicting the OSS evolution, because when there is a lack of analysis on OSS evolvability characteristics, it also becomes harder to predict its evolution.

- **Examining OSS evolution at software architecture level**

We have found that although an increasing amount of attention is being paid to the architecture of software systems due to its recognized role in fulfilling the quality requirements of a system [55], only few papers address OSS evolution at architectural level. Software evolution can be examined at different levels such as architectural level, detailed design and source code level. We have noticed from the review that most papers address OSS evolution at source code level. However, software architectures are inevitably subject to evolution. They expose the dimensions along which a system is expected to evolve [74], and provide basis for software evolution [126]. Therefore, it is of major importance to put more focus on managing OSS evolution and assessing OSS evolvability at the software architecture level besides the code-level evolution.

# Chapter 7. Validity Discussions

In general, our software architecture evolution research in this thesis is based on empirical studies. The software evolvability model, the formulation of the qualitative and quantitative evolvability analysis methods are built upon our systematic review of software architecture evolution research [35], our observations and experiences of working with many different types of industrial systems from different domains, several workshop discussions [32] [33] [34], and involvement of practitioners in the discussions.

In this chapter, we discuss the validity aspects of the research results described in the previous chapters. Because the ways for the data collection and research design vary for each research result we achieved, we go through each research contribution, and describe respective type of the validation used.

## 7.1 Validity Aspects on Software Evolvability Model

The formulation of the software evolvability model was based on multiple sources of evidence, including critical analysis of the existing literature and industrial case studies [31] [33]. We collected and analyzed data from published materials. The criteria on which literature to be 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.

Studying the existing quality models provided us the research idea of establishing a software evolvability model (see Chapter 4), and a basic idea on subcharacteristics that are essential for software evolvability (see Chapter 2). Moreover, based on our working experiences with various industrial software systems in different domains, we have found out particular quality attributes (subcharacteristics) that are essential for evolvability, i.e., analyzability, architectural integrity, portability, changeability and extensibility. Based on these evolvability subcharacteristics, we have

classified the set of quality characteristics (see Chapter 2.3) covered in the well-known quality models against evolvability subcharacteristics, as shown in Table 7-1.

**Table 7-1: Classification of quality characteristics in quality models**

| Classification | Quality Characteristics in Quality Models |
|---|---|
| Analyzability | Human Engineering (Boehm), Understandability (Boehm, ISO 9126) |
| Changeability | Flexibility (McCall), Modifiability (Boehm, ISO 9126) |
| Integrity | Reusability (McCall, Dromey) |
| Extensibility | Extensibility (FURPS) |
| Portability | Adaptability (FURPS, ISO 9126), Compatibility (FURPS), Interoperability (McCall, ISO 9126) |
| Testability | Correctness (McCall), Efficiency (McCall, Boehm, ISO 9126, Dromey) |

Apart from the development quality attributes that are explicitly addressed in the classification, the operational quality attributes, such as performance, reliability are also indirectly addressed in the sense that the improvement of these attributes are handled through e.g., analyzability and changeability. Portability and extensibility are explicit in the classification because they are essential for software evolvability. As a result, this classification is relevant for evolution of software-intensive systems, and covers the ranges of potential future changes that a software system may encounter during its life cycle.

The software evolvability model has been validated in various domains. The first industrial case study [31], in which we validated the evolvability model, is a representative and typical case which captures the commonplace situation of large complex software systems. From this case, we were convinced that there are also domain-specific attributes that are essential for software evolvability depending on the system's domain. After having outlined the software evolvability model based on the first industrial case [31] [33], we applied it also to other domains (see Chapter 5) that might have extended or different set of evolvability subcharacteristics. We further validated the software evolvability model in practice by studying also the documentation on architectural requirements and quality improvement requirements throughout the case studies. We also interviewed various stakeholders of different roles for their views on the set of abstractions on evolvability subcharacteristics.

.

## 7.2 Validity Aspects on AREA Process

The formulation of the general AREA process (see Chapter 4) was based on the qualitative and quantitative software evolvability analysis methods that we have developed. The process reflects the shared commonalities at the conceptual level.

- Qualitative evolvability analysis method

  The formulation of the qualitative software evolvability analysis method was based on multiple sources of evidence, including critical analysis of the existing literature (see Chapter 2 and Chapter 3), and an industrial case study [32] [33].

- Quantitative evolvability analysis method

  The formulation of the quantitative software evolvability analysis method was based on the experiences and lessons learned from applying the qualitative analysis method in an industrial case study (see Chapter 5), in which we discovered the potential to further extend the qualitative method with a quantitative feasibility in analyzing evolvability.

Based on Wohlin et al. [179], we will discuss below the validation of the applications of the software evolvability analysis methods in two large-scale industrial software systems in different domains (see Chapter 5). These case studies were in essence based on action research [8], i.e., the researchers participated in the process and perform empirical observations.

**Conclusion validity** [179] is concerned with the relationship between the treatment and the outcome. In the qualitative evolvability analysis, conclusion validity was addressed through:

- Architecture workshops with stakeholders to extract potential architectural requirements;
- Involvement of software architects and senior software developers in the analysis process to discuss candidate architectural solutions' impacts on evolvability;
- Researchers' experiences and involvement in the software product development.

In the quantitative evolvability analysis, as the answers to how important the evolvability subcharacteristics relate to each other is in the form of a subjective judgment, the answers tend not to be exactly the same for all participants, especially among stakeholders representing different roles. This was noticed in the case study, in which the preferences among stakeholder

roles differed whereas the architects had a commonly shared preference view on evolvability subcharacteristics. We saw this as a positive indication that there was an organizational alignment among architects. On the other hand, even the same participant might not provide exactly the same answer in terms of pair-wise comparison weights should the study be repeated. Therefore, the interviews were centered on asking a series of questions that were open-ended, i.e., conversational responses, to gain information about respective stakeholder's view and interpretation on evolvability subcharacteristics. This was to ensure that the stakeholders have well-elaborated and clarified understanding of evolvability subcharacteristics in their specific domain context that is required for providing meaningful pair-wise comparison weights for evolvability subcharacteristics and impacts of architectural alternatives on evolvability. Moreover, the calculation of consistency ratio in the AHP method also helped to check the consistency level of the individuals' answers.

I took part in applying the qualitative and quantitative evolvability analysis in both cases. All experiences are thus first-hand; in addition, other participants in the cases provided with material to make the conclusions less subjective. The risk of bias has been further decreased through the involvement of other researchers in the analysis of the experiences.

**Internal validity** [179] is concerned with the connection between the observed behavior and the proposed explanation for the behavior. In the quantitative evolvability analysis case, during the process of extracting stakeholders' preferences on evolvability subcharacteristics, a remark from the software designers was that a designer may have different valuation of evolvability subcharacteristics depending on the different subsystems that he/she has previously worked with. This is because different subsystems may have different quality attribute requirements in focus. Although we encouraged them to try to think at the system level, it may still become a threat to the study when extracting software designers' preferences of evolvability subcharacteristics. However, in the qualitative analysis process, this threat was addressed through the architecture workshops in which all the stakeholders could discuss about their perception and prioritization of architectural requirements together, and thus, could reach consensus.

**Construct validity** [179] is concerned with the relation between theory and observation. In both the qualitative and quantitative analysis cases, we informed the participants about the evolvability analysis process so that they became aware of the purpose and the intended results of the studies. Therefore, there is no threat of hypothesis guessing. However, one threat that might happen is in the qualitative analysis case when all the stakeholders

discussed at the architecture workshops about potential architectural requirements and their prioritization. Some people might not tell their true opinions that would deviate from the others. But, this type of threat was addressed in the quantitative analysis process in which separate interviews were conducted individually with respective stakeholders.

**External validity** [179] is concerned with generalization. Both the evolvability model and the AREA process are quite general, and can be applied for other properties (or at least for the properties of similar character as evolvability). The model described in Figure 4-1 is sufficiently general as it is just a matter of selecting subcharacteristics; the same is with AREA process. In some other system types, it may happen that subcharacteristic might be different, but the model and the analysis procedure is still valid. In both the qualitative and quantitative cases, the participants represented the different roles of stakeholders that are involved in software development. Therefore, there is no threat in the selection of participants. In addition, based on our experiences in both case studies, although the systems belonged to different domains – automation and telecommunication domains, the evolvability analysis methods seemed to be generally applicable. However, one threat to external validity is that there are some similarities between the two cases, such as large, complex, long-lived software-intensive systems with strong requirements for backward compatibility and no evolution breaks. Another threat is that both companies are large international ones though located in Sweden, and thus might impose some social and cultural behavior of people, especially during interviews and workshops.

## 7.3 Validity Aspects on Architecting for Software Evolvability

The systematic literature review of software architecture evolution research (see Chapter 3) was based on a formalized and repeatable process to document relevant knowledge on architecting for evolvability. All available research related to the research questions were thoroughly assessed and interpreted to answer the research questions as specified in Chapter 3.

The main threats to validity in this systematic review are bias in our selection of the studies to be included, and data extraction. To be able to identify relevant studies and ensure that the process of selection was unbiased, a research protocol was developed to define research questions, inclusion and exclusion criteria, and search strategy. The review protocol

was prepared by me, and was then reviewed by two other researchers to check the formulation of research questions, whether the search strings were appropriately derived from the research questions, and whether the data to be extracted would address the research questions. The review protocol was also reviewed by an external senior researcher from academia, who is experienced in systematic review within the research group. In addition, an earlier version of the paper was presented at an internal workshop within the research group for additional feedbacks, especially on the inclusion and exclusion criteria. For instance, in the beginning, we focused mainly on research papers and excluded experience reports. However, one comment from the workshop was that we also need to look into the experience reports to obtain a good understanding of the maturity and applicability of the approaches regarding the analysis and achievement of software evolvability at the architectural level. These comments were then taken into consideration when we started working on this systematic literature review. The external senior researcher and the participants at the internal workshop were all from academia.

Although the research protocol was reviewed by several senior researchers for feedback and was modified based on their comments to reduce the bias of the formalization of the protocol, due to our choice of search terms, there is still a risk that we might have missed some relevant studies, especially in cases when some software engineering keywords are not standardized and clearly defined, such as definitions for various quality attributes. We dealt with this threat by making sure that all the researchers participating in this review had the same definition in case of unclear terms, though in some cases it was hard to know how the authors of the reviewed papers defined for example adaptability or evolvability.

To further ensure the unbiased selection of articles, we performed a multi-step selection process to minimize the risk of exclusion of relevant studies. Three researchers were involved in the steps that concerned excluding studies based on the exclusion criteria as well as excluding irrelevant studies based on their titles and abstracts. We reviewed all the papers' titles and abstracts, and recorded independently the decisions if a paper would be selected for the full-text screening step. Afterwards, to ensure the reliability of inclusion decisions, we applied the Fleiss Kappa statistic [68] to measure the agreement among us three researchers. The initial value of the Kappa statistics was 0.64 which is within the range for significant agreement. Applying the Fleiss Kappa method gave us very good input on papers that we had discrepancies on, and thus, resulted in further discussions. Consequently, each discrepancy was discussed and resolved, and thus we

had full agreement on studies that should be included for the final full-text screening step. Throughout this selection process with discussions on potential primary studies' actual relevance, we had obtained a clear view on how to judge a paper's actual relevance for being included as a primary study. Therefore, we decided that I would take the lead in the full-text screening step, and facilitate the discussions that lead to the final paper selection for this review. Besides, additional reference checking of the identified studies was conducted to guarantee a representative set of studies for the review.

To ensure correctness in data extraction, we defined a data extraction form (see Table 3-2) to obtain consistent extraction of relevant information for answering the research questions. In addition, we performed quality assessment on relevant studies to ensure that the identified findings and implications came from a credible basis.

## 7.4 Validity Aspects on Open Source Software Evolution

The systematic literature review of open source software evolution (see Chapter 6) was based on a formalized and repeatable process to document relevant knowledge on open source software evolution. All available research related to the research questions were thoroughly assessed and interpreted to answer the research questions as specified in Chapter 6. The following types of validity issues were considered when interpreting the results from this review.

**Conclusion validity** [179] refers to the statistically significant relationship between the treatment and the outcome. One possible threat to conclusion validity is bias in data extraction. This was addressed through defining a data extraction form to ensure consistent extraction of relevant data to answering the research questions. The findings and implications were based on the extracted data.

**Internal validity** [179] 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. It is a concern for causal or explanatory studies. One possible threat to internal validity is the selection bias. We addressed this threat during the selection step of the review, i.e., the studies included in this review were identified through a thorough selection process which comprised of multiple stages. In the first stage, I and another

researcher independently selected and reviewed relevant papers from the complete set of papers retrieved on basis of the search strings. Then the selected papers were aggregated. After the first set of selected papers was selected, a third senior researcher performed random check to validate if it was the right selection of papers.

**Construct validity** [179] 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 studies identified from the systematic review were accumulated from multiple literature databases covering relevant journals, proceedings and book chapters. One possible threat to construct validity is bias in the selection of publications. This was addressed through specifying a research protocol that defined the research questions and objectives of the study, inclusion and exclusion criteria, search strings that we intended to use, the search strategy and strategy for data extraction. The research protocol and the identified publications were reviewed by several researchers to minimize the risk of exclusion of relevant studies. Besides, additional reference checking of the identified studies was conducted to guarantee a representative set of studies for the review.

# Chapter 8. Conclusions and Future Work

This chapter presents the conclusions of this thesis. First, the answers are presented to the research questions posed by the thesis. This is followed by a description of the research contributions. The chapter concludes with an outlook into future research directions.

## 8.1 Research Questions and Answers

In Chapter 1.3, the research questions of the thesis were presented. For each of these questions, we present a short answer here based on the work presented in the previous chapters.

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

Based on the definition in [154], the analysis of existing quality models (see Chapter 2), the analysis of the software quality challenges and assessment [67], the types of change stimuli and evolution [50], the taxonomy of software change based on various dimensions that characterize or influence the mechanisms of change [39], and experiences we gained in industrial case studies (see Chapter 4 and Chapter 5), we have discovered that only having a collection of the subcharacteristics of maintainability as defined in the ISO software quality standard [89] (see Chapter 2) is not sufficient for a software system to be evolvable. Therefore, we have (i) complimented and identified subcharacteristics that are of primary importance for an evolvable software system, and (ii) outlined a software evolvability model that provides a basis for analyzing and evaluating software evolvability.

The idea with the software 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.

The subcharacteristics that are of primary importance for software evolvability in a given context (long-lived software-intensive systems) are analyzability, architectural integrity, changeability, extensibility, portability, testability and domain-specific attributes. The validation of the software evolvability model along with identified evolvability subcharacteristics are described in Chapter 5, detailing the industrial case studies in automation domain and mobile network domain.

**Question 2: How to assess software evolvability of long-lived proprietary systems in a systematic manner?**

To be able to understand and analyze systematically the evolution of software system architectures, we have proposed (see Chapter 4) an architecture evolvability analysis (AREA) process that comprises of the following main activities:

- Elicit architectural concerns

  This activity extracts architectural concerns with respect to evolvability subcharacteristics among stakeholders either qualitatively or quantitatively.

- Analyze implications of change stimuli

  This activity analyzes the architecture for evolution, and identifies the impact of change stimuli on the current architecture. Accordingly, this activity focuses on discovering the problems the software architecture needs to solve, examining change stimuli and architectural concerns in order to obtain a set of potential architectural requirements.

- Propose architectural solutions

  This activity proposes candidate architecture solutions to accommodate to a set of potential architectural requirements.

- Assess architectural solutions

  This activity ensures that the architectural design decisions made are appropriate for software architecture evolution. The candidate architectural solutions are assessed against evolvability subcharacteristics, i.e., the implications of the potential architectural strategies and evolution path of the software architecture are analyzed either qualitatively or quantitatively.

The proposed AREA process provides repeatable techniques for performing the activities to support software architecture evolution. The activities are embedded in:

- A structured qualitative method (see Chapter 4.3) for analyzing evolvability at the architectural level;
- A quantitative evolvability analysis method (see Chapter 4.4) with explicit and quantitative treatment of stakeholders' evolvability concerns and potential architectural solutions' impacts on software evolvability.

Moreover, the qualitative and quantitative assessments manifested in the evolvability analysis process have been validated through their applications in two large-scale industrial software systems at ABB and Ericsson (see Chapter 5).

**Question 3: How is software evolvability addressed in the development and evolution of open source software?**

We have performed a systematic review (see Chapter 6) that comprises of 41 identified primary studies. Based on the research topics of these studies, we have classified them into four main categories of themes: software trends and patterns, evolution process support, evolvability characteristics, and examining OSS at software architecture level. The first category is further refined into three sub-categories: software growth, software maintenance and evolution economics, and prediction of software evolution. The main findings from this systematic review are:

- Most papers focus on using different metrics to analyze OSS evolution over time. Few papers have looked into the economic perspective, e.g., maintenance effort, and few papers utilize the historical evolution data for prediction of OSS evolution and development.
- Several evolution process support, different aspects that appear to have impact on the OSS evolution process are covered; these aspect include commenting practice, OSS evolution and maintenance evaluation model, structures and quality characteristics of resources such as repositories, mails, bug tracking systems, as well as tools that support data retrieval for evolution analysis.
- Determinism, understandability, modularity and complexity are related evolvability characteristics covered in the primary studies. However, there are more evolvability characteristics that are not covered such as changeability, extensibility, and testability. This also explains the findings in the analysis of OSS evolution trends category that most studies focus on the evolution history instead of predicting the OSS evolution, because when there is a lack of analysis on OSS evolvability characteristics, it also becomes harder to predict its evolution.

- Few papers address OSS evolution at architectural level. Most papers address OSS evolution at source code level.

## 8.2 Contributions

The main focus of this thesis is software evolvability analysis of proprietary systems. A supplementary research area is open source software evolution. This section summarizes our research contributions.

### 8.2.1 Main Research Contributions

The main contributions of the thesis are concerned with the software evolvability analysis of proprietary systems, and are summarized as follows:

- **Software evolvability model**

In this thesis, we have proposed a software evolvability model that provides a basis for analyzing software evolvability. This model refines software evolvability into a collection of subcharacteristics that can be measured through a number of measuring attributes, and is established as a first step towards analyzing and quantifying evolvability. The evolvability subcharacteristics are used as check points for evolvability evaluation and improvement. The software evolvability model has been validated in several industrial settings.

- **Software architecture evolvability analysis (AREA) process**

In this thesis, we have defined the software architecture evolvability analysis process (AREA) which engages stakeholders throughout the system development and evolution lifecycle to discover the driving architectural requirements, stakeholders' evolvability concerns, and potential architectural solutions' impact on evolvability of a software system. The analysis process can be carried out at many points during a system's lifecycle, and is stakeholder-focused.

The results of the evolvability analysis process include: (i) the prioritized architectural requirements; (ii) stakeholders' evolvability concerns; (iii) candidate architectural solutions; and (iv) the architectural solutions' impact on evolvability.

It is a challenging task for an architect to choose among competing candidate architectural solutions and ensure that the system constructed from the architecture satisfies its stakeholders' needs. Therefore, the results from the

evolvability analysis process are useful for an architect to design and evolve the architecture. The AREA process provides two repeatable techniques to understand and support software architecture evolution:

- **Qualitative evolvability analysis method**

  We have proposed a qualitative evolvability analysis method that focuses on improving the capability of being able to understand and analyze systematically the impact of change stimuli on software architecture evolution.

- **Quantitative evolvability analysis method**

  We have also proposed a quantitative evolvability analysis method that provides quantifications of stakeholders' evolvability concerns and potential architectural solutions' impacts on evolvability.

The above techniques have been validated through our participation in two large-scale industrial projects (at ABB and Ericsson) driven by the need of improving software evolvability. Based on our experiences, both the qualitative and quantitative analysis methods can be used as an integral part of software development and evolution process. Throughout the process of evolvability analysis at ABB, the architecture requirements and corresponding design decisions for the transition of architecture became more explicit, better founded and documented. The resulting analysis results were well accepted by the stakeholders involved in the analysis process, and became a blueprint for further implementation improvement. Throughout the process of evolvability analysis at Ericsson, the importance of various quality attributes perceived among different stakeholders was quantified and became more explicit. This quantification also served as a communication vehicle for further discussions among stakeholders. In both cases, by analyzing architectural improvement proposals with respect to their implications on evolvability subcharacteristics, we further avoided an ad hoc choice of potential evolution paths of software architecture.

- **Systematic review in architecting for software evolvability, revealing suggestions for further research and practice**

In this thesis, we have performed a systematic literature review of the existing studies in analyzing and achieving software evolvability at architectural level. These studies cover a spectrum of approaches with specific perspective or focus on a particular architecture-centric activity in software lifecycle, and belong to five main categories of themes:

a) Quality consideration during software architecture design
b) Architectural quality evaluation

c) Economic valuation
d) Architectural knowledge management
e) Modeling techniques.

A comprehensive overview and analysis of these categories and related studies is presented, as well as the implications for research and practitioners.

### 8.2.2 Supplementary Research Contribution

A supplementary research contribution is concerned with the open source software evolution, and is summarized below:

- **Systematic review in open source software evolution, revealing suggestions for further research**

In this thesis, we have also performed a systematic review of the existing studies in open source software evolution. These studies are grouped into four main categories of themes:

a) Software trends and patterns
b) Evolution process support
c) Evolvability characteristics addressed in OSS evolution
d) Examining OSS at software architecture level

A comprehensive overview and analysis of these categories and related studies is presented, describing how software evolvability is addressed during development and evolution of OSS, and identifying challenges and future research directions in OSS evolution.

## 8.3 Future Research Directions

A number of potential tracks for future research are identified as follows:

- **Further validation of evolvability analysis methods**

Although the software evolvability analysis methods developed in this research have been verified through industrial case studies of different domains in two different companies, one limitation is that there are some similarities between the two cases, such as large, complex, long-lived software-intensive systems with strong requirements for backward compatibility and no evolution breaks. Another limitation is that both

companies are large international ones though located in Sweden, and thus might impose some social and cultural behavior of people, especially during interviews and workshops. Therefore, future research includes additional validation and adaptation of the methods using multiple case studies in systems and cultures of different characteristics.

- **Further development of foundation theories**

There is a space to develop new foundation theories beyond Lehman's law, e.g., quantitative expression of evolvability, along with its measurement, monitoring, prediction, impact analysis, with practical value to software architecture evolution.

- **Novel methods to support ultra-large-systems evolution**

Considering that all artefacts produced and used during the entire software lifecycle are subject to changes, novel methods and tools need to be developed to be able to design ultra-large-systems that integrate and orchestrate the evolution of thousands of platforms, decision nodes, organizations and processes.

- **Further research in open source software (OSS) evolution**

There are three potential aspects for future research:

  - **Economic perspective and prediction of OSS evolution**

  Based on the systematic review that we performed, we have found that few studies have looked into the economic perspective, e.g., maintenance effort, and few papers utilize the historical evolution data for prediction of OSS evolution and development. Therefore, future research include examining OSS evolution from economic perspective as well as predicting OSS evolution based on historical evolution data.

  - **Evolvability characteristics of OSS**

  Based on the systematic review that we performed, we have found that some evolvability characteristics are not addressed in OSS evolution such as changeability, extensibility, and testability. This might also explain the findings that most studies focus on the evolution history instead of predicting the OSS evolution, because when there is a lack of analysis on OSS evolvability characteristics, it also becomes harder to predict its evolution. Therefore, future research includes further validation and adaptation of the proposed software evolvability model by applying it to open source software evolution.

- **Architecture level evolvability analysis of OSS evolution**

Based on the systematic review that we performed, we have found that only few studies address open source software (OSS) evolution at architectural level. Therefore, future research includes (i) putting more focus on managing OSS evolution, and assessing OSS evolvability at the software architecture level; and (ii) further validation and adaptation of the proposed evolvability analysis process and methods by applying to open source software evolution.

# Appendix A: Primary Studies in Chapter 3


[S1] T. Al-Naeem, J. Gorton, M. Ali Babar, F. Rabhi, B. Benatallah, A quality-driven systematic approach for architecting distributed software applications, International Conference on Software Engineering (ICSE), 2005.

[S2] M. Ali Babar, I. Gorton, A tool for managing software architecture knowledge, International Conference on Software Engineering Workshop on Sharing and Reusing architectural Knowledge-Architecture, Rationale, and Design Intent, 2007.

[S3] M. Ali Babar, I. Gorton, R. Jeffery, Capturing and using software architecture knowledge for architecture-based software development, International Conference on Quality Software (QSIC), pp. 169-176, 2005.

[S4] S. Anwar, M. Ramzan, A. Rauf, A. Ali Shahid, Software maintenance prediction using weighted scenarios: an architecture perspective, International Conference on Information Science and Applications (ICISA), 2010.

[S5] M. Aoyama, Continuous and discontinuous software evolution: aspects of software evolution across multiple product lines, International Conference on Software Engineering Workshop on Principles of Software Evolution, pp. 87-90, 2001.

[S6] R. Bahsoon, W. Emmerich, ArchOptions: a real options-based model for predicting the stability of software architectures, International Conference on Software Engineering Workshop on Economic-Driven Software Engineering Research, 2003.

[S7] R. Bahsoon, W. Emmerich, Evaluating architectural stability with real options theory, International Conference on Software Maintenance, pp. 443-447, 2004.

[S8] L. Bass, P. Clements, R. Kazman, Software architecture in practice, ISBN 0321154959, Addison-Wesley Professional, 2003.

[S9] P. Bengtsson, J. Bosch, Architecture level prediction of software maintenance, European Conference on Software Maintenance and Reengineering (CSMR), pp. 139-147, 1999.

[S10] P. Bengtsson, J. Bosch, Scenario-based software architecture reengineering, International Conference on Software Reuse, pp. 308-317, 1998.

[S11] P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet, Architecture-level modifiability analysis (ALMA), Journal of Systems and Software, vol. 69, pp. 129-147, 2004.

[S12] S. Bhattacharya, D. E. Perry, Architecture assessment model for system evolution, Working IEEE/IFIP Conference on Software Architecture (WICSA), 2007.

[S13] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach, Addison-Wesley Professional, ISBN 0-201-67494-7, 2000.

[S14] E. Bouwers, A. van Deursen, A lightweight sanity check for implemented architectures, IEEE Software, vol. 27, 2010.

[S15] H. P. Breivold, I. Crnkovic, P. J. Eriksson, Analyzing Software Evolvability, Annual IEEE International Computer Software and Applications Conference (COMPSAC), 2008.

[S16] H. P. Breivold, I. Crnkovic, R. Land, M. Larsson, Analyzing software evolvability of an industrial automation control system: a case study, International Conference on Software Engineering Advances (ICSEA), 2008.

[S17] R. Brcina, M. Riebisch, Architecting for evolvability by means of traceability and features, International Conference on Automated Software Engineering (ASE) Workshops, pp. 72-81, 2008.

[S18] T. R. Browning, E. C. Honour, Measuring the life-cycle value of enduring systems, Journal of Systems Engineering, vol. 11, 2008.

[S19] W. Bu, A. Tang, J. Han, An analysis of decision-centric architectural design approaches, International Conference on Software Engineering Workshop on Sharing and Reusing Architectural Knowledge (SHARK), 2009.

[S20] J. E. Burge,D. C. Brown, Software Engineering Using RATionale, Journal of Systems & Software, vol. 81, pp. 395-413, 2008.

[S21] R. Capilla, F. Nava, J. C. Dueas, Modeling and documenting the evolution of architectural design decisions, International Conference on Software Engineering Workshop on Sharing and Reusing Architectural Knowledge-Architecture, Rationale, and Design Intent, 2007.

[S22] R. Capilla, F. Nava, S. Pérez, J. C. Dueñas, A web-based tool for managing architectural design decisions, ACM SIGSOFT Software Engineering Notes, vol. 31, 2006.

[S23] J. Carriere, R. Kazman, I. Ozkaya, A cost-benefit framework for making architectural decisions in a business context, International Conference on Software Engineering, 2010.

[S24] S. J. Carriere, R. Kazman, S. G. Woods, Assessing and maintaining architectural quality, European Conference on Software Maintenance and Reengineering, pp. 22-30, 1999.

[S25] H. B. Christensen, K. M. Hansen, An empirical investigation of architectural prototyping, Journal of Systems and Software, vol. 83, pp. 133-142, 2010.

[S26] L. Chung, K. Cooper, A. Yi, Developing adaptable software architectures using design patterns: an NFR approach, Computer Standards & Interfaces, vol. 25, pp. 253-260, 2003.

[S27] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos, Non-functional Requirements in Software Engineering, ISBN 978-0-7923-8666-7, Springer, 2000.

[S28] L. Chung, N. Subramanian, Process-oriented metrics for software architecture adaptability, IEEE International Symposium on Requirements Engineering, pp. 310-311, 2001.

[S29] P. Clements, L. Bass, Business goals as architectural knowledge, International Conference on Software Engineering Workshop on Sharing and Reusing Architectural Knowledge (SHARK), 2010.

[S30] P. Clements, R. Kazman, M. Klein, Evaluating Software Architectures: Methods and Case Studies, ISBN 0-201-70482-x, Addison-Wesley, 2006.

[S31] X. Cui, Y. Sun, S. Xiao, H. Mei, Architecture design for the large-scale software-intensive systems: a decision-oriented approach and the experience, International Conference on Engineering of Complex Computer Systems (ICECCS), 2009.

[S32] R. C. de Boer, P. Lago, A. Telea, H. van Vliet, Ontology-driven visualization of architectural design decisions, Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA), 2009.

[S33] C. Del Rosso, Continuous evolution through software architecture evaluation: A case study, Journal of Software Maintenance and Evolution, vol. 18, pp. 351-383, 2006.

[S34] C. Del Rosso, A. Maccari, Assessing the architectonics of large, software-intensive systems using a knowledge-based approach, Working IEEE/IFIP Conference on Software Architecture (WICSA), 2007.

[S35] A. Engel, T. R. Browning, Designing systems for adaptability by means of architecture options, Journal of Systems Engineering, vol. 11, 2008.

[S36] R. Farenhorst, R. Izaks, P. Lago, H. van Vliet, A Just-In-Time Architectural Knowledge Sharing Portal, Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 125-134, 2008.

[S37] E. Fricke, B. Gebhard, H. Negele, E. Igenbergs, Coping with changes: Causes, findings, and strategies, Journal of Systems Engineering, vol. 3, pp. 169-179, 2000.

[S38] E. Fricke, A. P. Schulz, Design for changeability (DfC): Principles to enable changes in systems throughout their entire lifecycle, Journal of Systems Engineering, vol. 8, 2005.

[S39] D. Garlan, J. M. Barnes, B. Schmerl, O. Celiku, Evolution styles: foundations and tool support for software architecture evolution, Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, 2009.

[S40] S. Giesecke, W. Hasselbring, M. Riebisch, Classifying architectural constraints as a basis for software quality assessment, Advanced Engineering Informatics, vol. 21, pp. 169-179, 2007.

[S41] M. O. Hassan, L. Deruelle, H. Basson, A knowledge-based system for change impact analysis on software architecture, International Conference on Research Challenges in Information Science, 2010.

[S42] C. Hofmeister, R. Nord, D. Soni, Applied Software Architecture: A Practical Guide for Software Designers, ISBN 0201325713, Addison-Wesley Professional, 2000.

[S43] A. Jansen, P. Avgeriou, J. S. van der Ven, Enriching software architecture documentation, Journal of Systems and Software, vol. 82, pp. 1232-1248, 2009.

[S44] A. Jansen, J. Bosch, P. Avgeriou, Documenting after the fact: Recovering architectural design decisions, Journal of Systems & Software, vol. 81, pp. 536-557, 2008.

[S45] A. Jansen, J. Van der Ven, P. Avgeriou, D. K. Hammer, Tool support for architectural decisions, Working IEEE/IFIP Conference on Software Architecture (WICSA) 2007.

[S46] R. Kazman, J. Asundi, M. Klein, Quantifying the costs and benefits of architectural decisions, International Conference on Software Engineering 2001.

[S47] R. Kazman, L. Bass, G. Abowd, M. Webb, SAAM: a method for analyzing the properties of software architectures, International Conference on Software Engineering, pp. 81-90, 1994.

[S48] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, The architecture tradeoff analysis method, IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 68-78, 1998.

[S49] S. Kim, D. K. Kim, L. Lu, S. Park, Quality-driven architecture development using architectural tactics, Journal of Systems and Software, vol. 82, pp. 1211-1231, 2009.

[S50] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, H. Lipson, Attribute-based architecture styles, Working IEEE/IFIP Conference on Software Architecture (WICSA) 1999.

[S51] P. Lago, H. Muccini, H. van Vliet, A scoped approach to traceability management, Journal of Systems and Software, vol. 82, pp. 168-182, 2009.

[S52] P. Lago, H. van Vliet, Explicit assumptions enrich architectural models, International Conference on Software Engineering, pp. 206-214, 2005.

[S53] N. Lassing, P. Bengtsson, H. van Vliet, J. Bosch, Experiences with ALMA: Architecture-Level Modifiability Analysis, Journal of Systems and Software, vol. 61, pp. 47-57, 2002.

[S54] N. Lassing, D. Rijsenbrij, H. van Vliet, How well can we predict changes at architecture design time?, Journal of Systems and Software, vol. 65, pp. 141-153, 2003.

[S55] J. Lee, D. H. Lee, Quantitative tradeoff analysis of software architecture using the architecture analysis and design language, ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009.

[S56] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, W. M. Turski, Metrics and laws of software evolution-the nineties view, 4th International Symposium on Software Metrics, 1997.

[S57] X. Liu, Q. Wang, Study on application of a quantitative evaluation approach for software architecture adaptability, International Conference on Quality Software (QSIC), pp. 265-272, 2005.

[S58] C. H. Lung, S. Bot, K. Kalaichelvan, R. Kazman, An approach to software architecture analysis for evolution and reusability, Conference of the Centre for Advanced Studies on Collaborative Research, IBM Center for Advanced Studies Conference, 1997.

[S59] T. Marew, J.S. Lee, D. H. Bae, Tactics based approach for integrating non-functional requirements in object-oriented analysis and design, Journal of Systems and Software, vol. 82, pp. 1642-1656, 2009.

[S60] A. Mubin, D. Ray, R. Rahman, Architecting an evolvable system by iterative object-process modeling, World Congress on Computer Science and Information Engineering (CSIE), 2009.

[S61] E. C. Nistor, A. van der Hoek, Explicit concern-driven development with ArchEvol, IEEE/ACM International Conference on Advanced Software Engineering, 2009.

[S62] F. G. Olumofin, V. B. Misic, A holistic architecture assessment method for software product lines, Information and Software Technology, vol. 49, pp. 309-323, 2007.

[S63] E. Ovaska, A. Evesti, K. Henttonen, M. Palviainen, P. Aho, Knowledge based quality-driven architecture design and evaluation, Journal of Information and Software Technology, vol 52, pp.577-601, 2010.

[S64] I. Ozkaya, R. Kazman, M. Klein, Quality-Attribute Based Economic Valuation of Architectural Patterns, International Workshop on the Economics of Software and Computation, 2007.

[S65] C. Pahl, S. Giesecke, W. Hasselbring, Ontology-based modeling of architectural styles, Journal of Information and Software Technology, vol. 51, pp. 1739-1749, 2009.

[S66] D. Port, L. Huang. Strategic architectural flexibility, International Conference on Software Maintenance (ICSM), pp. 389-396, 2003.

[S67] J. F. Ramil, M. M. Lehman, Metrics of software evolution as effort predictors-a case study, International Conference on Software Maintenance (ICSM), pp. 163-172, 2000.

[S68] R. Roeller, P. Lago, H. van Vliet, Recovering architectural assumptions, Journal of Systems and Software, vol. 79, pp. 552-573, 2006.

[S69] G. Scanniello, A. D'Amico, C. D'Amico, T. D'Amico, Architectural layer recovery for software system understanding and evolution, Software: Practice and Experience, vol. 40, pp. 897-916, 2010.

[S70] M. Shahin, P. Liang, M. Reza, Improving understandability of architecture design through visualization of architectural design decision, International Conference on Software Engineering Workshop on Sharing and Reusing Architectural Knowledge (SHARK), 2010.

[S71] N. Subramanian, L. Chung, Process-oriented metrics for software architecture evolvability, International Workshop on Principles of Software Evolution, pp. 65-70, 2003.

[S72] K. J. Sullivan, W. G. Griswold, Y. Cai, B. Hallen, The structure and value of modularity in software design, European Software Engineering Conference held jointly with 9th ACM SIGSOFT

International Symposium on Foundations of Software Engineering, pp. 99-108, 2001.

[S73] M. Svahnberg, An industrial study on building consensus around software architectures and quality attributes, Information and Software Technology, vol. 46, pp. 805-818, 2004.

[S74] L. Tahvildari, K. Kontogiannis, J. Mylopoulos, Quality-driven software re-engineering, Journal of Systems and Software, vol. 66, pp. 225-239, 2003.

[S75] T. Tamai, Y. Torimitsu, Software lifetime and its evolution process over generations, International Conference on Software Maintenance, pp. 63-69, 1992.

[S76] D. Tamzalit, T. Mens, Guiding architectural restructuring through architectural styles, International Conference and Workshops on Engineering of Computer-Based Systems (ECBS), 2010.

[S77] A. Tang, M. Ali Babar, I. Gorton, J. Han, A survey of architecture design rationale, Journal of Systems and Software, vol. 79, pp. 1792-1804, 2006.

[S78] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, M. Ali-Babar, A Comparative Study of Architecture Knowledge Management Tools, Journal of Systems and Software, vol. 83, pp. 352-370, 2009.

[S79] P. Tarvainen, Adaptability evaluation of software architectures; A case study, Annual International Computer Software and Applications Conference (COMPSAC), pp. 579-584, 2007.

[S80] M. van den Berg, A. Tang, R. Farenhorst, A constraint-oriented approach to software architecture design, International Conference on Quality Software, 2009.

[S81] W. M. N. Wan-Kadir, P. Loucopoulos, Relating evolving business rules to software design, Journal of Systems Architecture, vol. 50, pp. 367-382, 2004.

[S82] L. Zhu, M. Ali Babar, R. Jeffery, Mining patterns to support software architecture evaluation, Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 25-34, 2004.

# Appendix B: Primary Studies in Chapter 6


[S1] A. Al-Ajlan, The evolution of open source software using eclipse metrics, International Conference on New Trends in Information and Service Science (NISS), 2009.

[S2] S. Ali, O. Maqbool, Monitoring software evolution using multiple types of changes, International Conference on Emerging Technologies (ICET), 2009.

[S3] O. Arafat, D. Riehle, The commenting practice of open source, 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA), 2009.

[S4] U. Asklund, L. Bendix, A study of configuration management in open source software projects, IEE Proceedings of Software, Issue 1, pp. 40-46, 2002.

[S5] A. Bachmann, A. Bernstein, Software process data quality and characteristics – a historical view on open and closed source projects, Proceedings of the joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, 2009.

[S6] A. Bauer, M. Pizka, The contribution of free software to software evolution, International Workshop on Principles of Software Evolution (IWPSE), 2003.

[S7] K. Beecher, A. Capiluppi, C. Boldyreff, Identifying exogenous drivers and evolutionary stages in FLOSS projects, Journal of Systems and Software, vol. 82, issue 5, pp. 739-750, 2009.

[S8] S. Bouktif, G. Antoniol, E. Merlo, A feedback based quality assessment to support open source software evolution: the GRASS case study, International Conference on Software Maintenance (ICSM), 2006.

[S9] A. Capiluppi, Models for the evolution of OS projects, International Conference on Software Maintenance (ICSM), 2003.

[S10] A. Capiluppi, K. Beecher, Structural complexity and decay in FLOSS systems: an inter-repository study, European Conference on Software Maintenance and Reengineering, 2009.

[S11] A. Capiluppi, A. E. Faria, J. F. Ramil, Exploring the relationship between cumulative change and complexity in an open source system, European Conference on Software Maintenance and Reengineering (CSMR), 2005.

[S12] A. Capiluppi, J. M. Gonzalez-Barahona, I. Herraiz, G. Robles, Adapting the "staged model for software evolution" to free/libre/open source software, International Workshop on Principles of Software Evolution (IWPSE) in conjunction with ESEC/FSE joint meeting, 2007.

[S13] A. Capiluppi, M. Morisio, P. Lago, Evolution of understandability in OSS projects, European Conference on Software Maintenance and Reengineering (CSMR), 2004.

[S14] A. Capiluppi, M. Morisio, J. F. Ramil, Structural evolution of an open source system: a case study, International Workshop on Program Comprehension (IWPC), 2004.

[S15] A. Capiluppi, J. F. Ramil, Studying the evolution of open source systems at different levels of granularity: two case studies, International Workshop on Principles of Software Evolution, 2004.

[S16] E. Capra, Mining open source web repositories to measure the cost of evolutionary reuse, International Conference on Digital Information Management, 2006.

[S17] E. Capra, C. Francalanci, F. Merlo, The economics of open source software: an empirical analysis of maintenance costs, International Conference on Software Maintenance (ICSM), 2007.

[S18] C. A. Conley, L. Sproull, Easier said than done: an empirical investigation of software design and quality in open source software development, 42nd Hawaii International Conference on System Sciences, 2008.

[S19] J.C. Deprez, F. F. Monfils, M. Ciolkowski, M. Soto, Defining software evolvability from a free/open-source software perspective, International IEEE Workshop on Software Evolvability, 2007.

[S20] M. W. Godfrey, Q. Tu, Evolution in open source software: a case study, International Conference on Software Maintenance, 2000.

[S21] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. José Amor, D. M. German, Macro-level software evolution: a case study of a large software compilation, Journal of Empirical Software Engineering, vol. 14(3), 2007.

[S22] I. Harraiz, J. M. Gonzalez-Barahona, G. Robles, D. M. German, On the prediction of the evolution of libre software projects, International Conference on Software Maintenance (ICSM), 2007.

[S23] I. Herraiz, J. M. Gonzalez-Barahona, G. Robles, Determinism and evolution, International Working Conference on Mining Software Repositories (MSR), 2008.

[S24] C. Izurieta, J. Bieman, The evolution of FreeBSD and Linux, International Symposium on Empirical Software Engineering (ISESE), 2006.

[S25] S. Koch, Software evolution in open source projects – a large-scale investigation, Journal of Software Maintenance and Evolution: Research and Practice, vol. 19, issue 6, pp. 361-382, 2007.

[S26] T. Koponen, Evaluation framework for open source software maintenance, International Conference on Software Engineering Advances (ICSEA), 2006.

[S27] E. Y. Nakagawa, E. P. M. de Sousa, K. de Brito Murata, G. de Faria Andery, et al., Software architecture relevance in open source software evolution: a case study, 32nd Annual IEEE International Computer Software and Applications Conference, 2008.

[S28] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, Y. Ye, Evolution patterns of open-source software systems and communities, International Workshop on Principles of Software Evolution (IWPSE), 2002.

[S29] J. W. Paulson, G. Succi, A. Eberlein, An empirical study of open-source and closed-source software products, Journal of IEEE Transactions on Software Engineering, vol. 30, issue 4, 2004.

[S30] U. Raja, D. P. Hale, J. E. Hale, Modeling software evolution defects: a time series approach, Journal of Software Maintenance and Evolution: Research and Practice, vol. 21, issue 1, pp. 49-71, 2009.

[S31] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, I. Herraiz, Evolution and growth in large libre software projects, International Workshop on Principles of Software Evolution (IWPSE), 2005.

[S32] G. Robles, J. M. Gonzalez-Barahona, D. Izquierdo-Cortazar, B. E. Erlandson, Tools for the study of the usual data sources found in libre software projects, Software Applications: Concepts, Methodologies, Tools, and Applications, 2009.

[S33] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, J. J. Amor, Mining large software compilations over time: another perspective of software evolution, International Workshop on Mining Software Repositories (MSR), 2006.

[S34] M. Simmons, P. Vercellone-Smith, P. A. Laplante, Understanding open source software through software archaeology: the case of Nethack, 30th Annual IEEE/NASA Software Engineering Workshop (SEW), 2006.

[S35] N. Smith, A. Capiluppi, J. F. Ramil, A study of open source software evolution data using qualitative simulation, Journal of Software Maintenance and Evolution: Research and Practice, vol. 10, issue 3, pp. 287-300, 2005.

[S36] L. G. Thomas, S. R. Schach, G. Z. Heller, J. Offutt, Impact of release intervals on empirical research into software evolution, with application to the maintainability of Linux, IET Software, vol. 3, issue 1, pp. 58-66, 2009.

[S37] J. B. Tran, M. W. Godfrey, E. H. S. Lee, R. C. Holt, Architectural repair of open source software, International Workshop on Program Comprehension (IWPC), 2000.

[S38] Y. Wang, D. Guo, H. Shi, Measuring the evolution of open source software systems with their communities, ACM SIGSOFT Software Engineering Notes, vol. 32, issue 6, 2007.

[S39] R. van Wendel de Joode, T. M. Egyedi, Handling variety: the tension between adaptability and interoperability of open source software, Journal of Computer Standards & Interfaces, vol. 28, issue 1, pp. 109-221, 2005.

[S40] G. Xie, J. Chen, I. Neamtiu, Towards a better understanding of software evolution: an empirical study on open source software, International Conference on Software Maintenance, 2009.

[S41] L. Yu, Indirectly predicting the maintenance effort of open-source software, Journal of Software Maintenance and Evolution: Research and Practice, vol. 18, issue 5, pp. 311-332, 2006.

# References


[1] OSS/BSS reference architecture and its implementation scenario for fulfillment, *http://www.nokia.com/NOKIA_COM_1/About_Nokia/Press/White_Papers/pdf_files/nokia_tietoenator_0605_net.pdf*.

[2] A. Al-Ajlan, The evolution of open source software using Eclipse metrics, International Conference on New Trends in Information and Service Science (NISS), 2009.

[3] M. S. Ali, M. Ali Babar, L. Chen, K. J. Stol, A systematic review of comparative evidence of aspect-oriented programming, Information and Software Technology, vol. 52(9), pp. 871-887, 2010.

[4] S. Ali, O. Maqbool, Monitoring software evolution using multiple types of changes, International Conference on Emerging Technologies (ICET), 2009.

[5] B. Anda, K. Hansen, A case study on the application of UML in legacy development, the ACM/IEEE International Symposium on Empirical Software Engineering (ISESE), 2006.

[6] M. Aoyama, Metrics and analysis of software architecture evolution with discontinuity, International Workshop on Principles of Software Evolution (IWPSE), 2002.

[7] O. Arafat, D. Riehle, The commenting practice of open source, 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA), 2009.

[8] C. Argyris, R. Putnam, D. M. L. Smith, Action Science: Concepts, Methods, and Skills for Research and Intervention, Jossey-Bass Inc. Publications, 1985.

[9] R. S. Arnold, Software restructuring, Proceedings of the IEEE, vol. 77, issue 4, pp. 607-617, 1989.

[10] R. S. Arnold, Software Reengineering, ISBN 0818632712, IEEE Computer Society Press, 1993.

[11] U. Asklund, L. Bendix, A study of configuration management in open source software projects, IEE Proceedings of Software, vol. 149, issue 1, pp. 40-46, 2002.

[12] A. Avritzer, E. J. Weyuker, Metrics to assess the likelihood of project success based on architecture reviews, Journal of Empirical Software Engineering, vol. 4(3), pp. 199-215, 1999.

[13] A. Bachmann, A. Bernstein, Software process data quality and characteristics: a historical view on open and closed source projects, Proceedings of the joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, 2009.

[14] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, A. Vilbig, A meta-model for representing variability in product family development, Software Product-Family Engineering, Lecture Notes in Computer Science, vol. 3014, pp. 66-80, 2004.

[15] P. Baker, S. Loh, F. Weil, Model-driven engineering in a large industrial context - Motorola case study, Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 3713, pp. 476-491, 2005.

[16] C. Y. Baldwin, K. B. Clark, Design Rules, Vol. 1: The Power of Modularity, ISBN 0262024667, MIT Press Cambridge, 2000.

[17] O. Barais, E. Cariou, L. Duchien, N. Pessemier, L. Seinturier, TranSAT: a framework for the specifcation of software architecture evolution, Workshop on Coordination and Adaptation Techniques for Software Entities (ECOOP), 2004.

[18] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, ISBN 0321154959, Addison-Wesley Longman Publishing Co., Inc. 2003.

[19] J. Bayer, J. F. Girard, M. Wurthner, J. M. DeBaud, M. Apel, Transitioning legacy assets to a product line architecture, ESEC/FSE Software Engineering, Lecture Notes in Computer Science, vol. 1686, pp. 446-63, 1999.

[20] K. Beecher, A. Capiluppi, C. Boldyreff, Identifying exogenous drivers and evolutionary stages in FLOSS projects, Journal of Systems and Software, vol. 82, issue 5, pp. 739-750, 2009.

[21] K. H. Bennett, V. T. Rajlich, Software maintenance and evolution: a roadmap, ICSE Proceedings of the Conference on the Future of Software Engineering, 2000.

[22] K. Bennett, Software evolution: past, present and future, Journal of Information and Software Technology, vol. 38, issue 11, pp. 673-680, 1996.

[23] D. Benyon, P. Turner, S. Turner, Designing Interactive Systems: People, Activities, Contexts, Technologies, ISBN 0321116291, Addison-Wesley, 2005.

[24] B. W. Boehm, A spiral model of software development and enhancement, Computer, vol. 21, issue 5, pp. 61-72, 1988.

[25] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, M. J. Merritt, Characteristics of Software Quality, ISBN 0444851054, North-Holland Publishing Co., 1978.

[26] I. Borne, S. Demeyer, G. H. Galal, Object-oriented architectural evolution, Object-Oriented Technology ECOOP, Lecture Notes in Computer Science, vol. 1743, pp. 57-64, 1999.

[27] A. Boronat, Automatic reengineering in MDA using rewriting logic as transformation engine, 9th European Conference on Software Maintenance and Reengineering (CSMR), 2005.

[28] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach, ISBN 978-0-201-67494-1, Addison-Wesley, 2000.

[29] S. Bouktif, G. Antoniol, E. Merlo, A feedback based quality assessment to support open source software evolution: the grass case study, International Conference on Software Maintenance (ICSM), 2006.

[30] H. P. Breivold, Software Architecture Evolution and Software Evolvability, Licentiate Thesis, ISBN 978-91-86135-15-7, Mälardalen University, 2009.

[31] H. P. Breivold, I. Crnkovic, Using software evolvability model for evolvability analysis, ISSN 1404-3041, Mälardalen Real-Time Research Center, Mälardalen University, 2008.

[32] H. P. Breivold, I. Crnkovic, An extended quantitative analysis approach for architecting evolvable software systems, Computing Professionals Conference Workshop on Industrial Software Evolution and Maintenance Processes (WISEMP), 2010.

[33] H. P. Breivold, I. Crnkovic, P. J. Eriksson, Analyzing software evolvability, IEEE International Computer Software and Applications Conference (COMPSAC), 2008.

[34] H. P. Breivold, I. Crnkovic, R. Land, M. Larsson, Analyzing software evolvability of an industrial automation control system: a case study, International Conference on Software Engineering Advances (ICSEA), 2008.

[35] H. P. Breivold, I. Crnkovic, M. Larsson, A systematic review of software architecture evolution research, Journal of Information and Software Technology, doi: 10.1016/j.infsof.2011.06.002, 2011.

[36] H. P. Breivold, M. Larsson, Component-based and service-oriented software engineering: key concepts and principles, 33rd Euromicro Conference on Software Engineering and Advanced Applications, 2007.

[37] J. Brichau, R. Chitchyan, A. Garcia, A. Rashid, S. Clarke, E. D'Hondt, M. Haupt, W. Joosen, S. Katz, J. Noyé, A model curriculum for aspect-oriented software development, IEEE Software, vol. 23(6), pp. 53-61, 2006.

[38] F. P. Brooks, No Silver Bullet, IEEE Computer, vol. 20, pp. 10-19, 1987.

[39] J. Buckley, T. Mens, M. Zenger, A. Rashid, G. Kniesel, Towards a taxonomy of software change, Journal of Software Maintenance and Evolution: Research and Practice, vol. 17, issue 5, pp. 309-332, 2004.

[40] A. Capiluppi, Models for the evolution of OS projects, International Conference on Software Maintenance (ICSM), 2003.

[41] A. Capiluppi, K. Beecher, Structural complexity and decay in FLOSS systems: an inter-repository study, European Conference on Software Maintenance and Reengineering, 2009.

[42] A. Capiluppi, A. E. Faria, J. F. Ramil, Exploring the relationship between cumulative change and complexity in an open source system, European Conference on Software Maintenance and Reengineering (CSMR), 2005.

[43] A. Capiluppi, J. M. González-Barahona, I. Herraiz, G. Robles, Adapting the staged model for software evolution to free/libre/open source software, International Workshop on Principles of Software Evolution (IWPSE), 2007.

[44] A. Capiluppi, M. Morisio, P. Lago, Evolution of understandability in oss projects, European Conference on Software Maintenance and Reengineering (CSMR), 2004.

[45] A. Capiluppi, M. Morisio, J. F. Ramil, Structural evolution of an open source system: a case study, International Workshop on Program Comprehension (IWPC), 2004.

[46] A. Capiluppi, J. F. Ramil, Studying the evolution of open source systems at different levels of granularity: two case studies, International Workshop on Principles of Software Evolution (IWPSE), 2004.

[47] E. Capra, Mining open source web repositories to measure the cost of evolutionary reuse, International Conference on Digital Information Management (ICDIM), 2006.

[48] E. Capra, C. Francalanci, F. Merlo, The economics of open source software: an empirical analysis of maintenance costs, International Conference on Software Maintenance (ICSM), 2007.

[49] H. Cervantes, R. S. Hall, Autonomous adaptation to dynamic availability using a service-oriented component model, International Conference on Software Engineering (ICSE), 2004.

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

[51] C. Chavez, A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena, Crosscutting interfaces for aspect-oriented modeling, Journal of the Brazilian Computer Society, vol. 12(1), pp. 43-58, 2006.

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

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

[54] L. Chung, A. Nixon, E. Yu, J. Mylopoulos, Non-Functional Requirements in Software Engineering, ISBN 9780792386667, Kluwer Academic Publishers, 1999.

[55] P. Clements, R. Kazman, M. Klein, Evaluating Software Architectures: Methods and Case Studies, ISBN 0-201-70482-X, Addison-Wesley, 2002.

[56] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, ISBN-10 9780201703320, Addison-Wesley Professional, 2002.

[57] A. Cockburn, Agile Software Development, ISBN-10 0201699699, Addison-Wesley Professional, 2002.

[58] C. A. Conley, L. Sproull, Easier said than done: an empirical investigation of software design and quality in open source software development, 42nd Hawaii International Conference on System Sciences, 2008.

[59] J. O. Coplien, Multi-paradigm design for C++, ISBN: 0-201-82467-1, Addison-Wesley Longman Publishing Co, Inc, USA, 1999.

[60] S. Demeyer, S. Ducasse, O. M. Nierstrasz, Object-Oriented Reengineering Patterns, ISBN 978-3-9523341-2-6, Morgan Kaufmann, 2003.

[61] D. Dhungana, T. Neumayer, P. Grünbacher, R. Rabiser, Supporting evolution in model-based product line engineering, International Software Product Line Conference, 2008.

[62] R. G. Dromey, Cornering the Chimera, IEEE Software, vol. 13, pp. 33-43, 1996.

[63] J. Estublier, G. Vega, 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.

[64] D. Falessi, R. Capilla, G. Cantone, A value-based approach for documenting design decisions rationale: a replicated experiment, International Workshop on Sharing and Reusing Architectural Knowledge (SHARK), 2008.

[65] D. Faust, C. Verhoef, Software product line migration and deployment, Software: Practice and Experience, vol. 33, issue 10, pp. 933-955, 2003.

[66] N. Fenton, S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, ISBN 0534954251, PWS Publishing Co. Boston, 1997.

[67] R. Fitzpatrick, P. Smith, B. O'Shea, Software quality challenges, Workshop on Software Quality at the 26th International Conference on Software Engineering, 2004.

[68] J. L. Fleiss, Measuring nominal scale agreement among many raters, Psychological bulletin, vol. 76(5), pp. 378-382, 1971.

[69] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, J. M. Jézéquel, Model-driven engineering for software migration in a large industrial context, Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 4735, pp. 482-497, 2007.

[70] M. Fowler, Refactoring: Improving the Design of Existing Code, ISBN 0-201-48567-2, Addison-Wesley Professional, 1999.

[71] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, ISBN 0201633612, Addison-Wesley Professional Computing Series, 1995.

[72] A. Garcia, C. Chavez, T. Batista, C. Sant'Anna, U. Kulesza, A. Rashid, C. Lucena, On the modular representation of architectural aspects, Software Architecture, Lecture Notes in Computer Science, vol. 4344, 2006.

[73] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. von Staa, Modularizing design patterns with aspects: a quantitative study, Transactions on aspect-oriented software development I, Lecture Notes in Computer Science, vol. 3880, pp. 36-74, 2006.

[74] D. Garlan, Software architecture: a roadmap, ICSE Proceedings of the Conference on the Future of Software Engineering, 2000.

[75] D. Garlan, J. M. Barnes, B. Schmerl, O. Celiku, Evolution styles: foundations and tool support for software architecture evolution, Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA), 2009.

[76] L. R. Gay, G. E. Mills, P. W. Airasian, Educational Research: Competencies for Analysis and Applications, ISBN-10 9780131185340, Prentice Hall, 2005.

[77] C. Gibbs, C. R. Liu, Y. Coady, Sustainable system infrastructure and big bang evolution: can aspects keep pace?, ECOOP - Object-Oriented Programming, Lecture Notes in Computer Science, vol. 3586, 2005.

[78] T. Gilb, Evolutionary development [software], SIGSOFT Software Engineering Notes, vol. 6, p. 17, 1981.

[79] T. Gilb, The 10 most powerful principles for quality in software and software organizations, Cross-Talk, Nov, 2002.

[80] M. W. Godfrey, D. M. German, The past, present, and future of software evolution, Frontiers of Software Maintenance (FoSM), 2008.

[81] M. W. Godfrey, Q. Tu, Evolution in open source software: a case study, International Conference on Software Maintenance, 2000.

[82] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, D. M. German, Macro-level software evolution: a case study of a large software compilation, Journal of Empirical Software Engineering, vol. 14, pp. 262-285, 2009.

[83] R. B. Grady D. L. Caswell, Software Metrics: Establishing a Company-Wide Program, ISBN-10 9780138218447, Prentice-Hall, 1987.

[84] I. Herraiz, J. M. Gonzalez-Barahona, G. Robles, Determinism and evolution, International Working Conference on Mining Software Repositories (MSR), 2008.

[85] I. Herraiz, J. M. Gonzalez-Barahona, G. Robles, D. M. German, On the prediction of the evolution of libre software projects, International Conference on Software Maintenance (ICSM), 2007.

[86] I. Herraiz, G. Robles, J. M. Gonzalez-Barahona, A. Capiluppi, J. F. Ramil, Comparison between SLOCs and number of files as size metrics for software evolution analysis, Conference on Software Maintenance and Reengineering (CSMR), 2006.

[87] H. J. Holz, A. Applin, B. Haberman, D. Joyce, H. Purchase, C. Reed, Research methods in computing: what are they, and how

should we teach them?, Annual Joint Conference Integrating Technology into Computer Science Education, 2006.

[88] IEEE-1471, IEEE Recommended Practices for Architectural Description of Software-Intensive Systems, 2000.

[89] ISO9126, ISO/IEC 9126-1, International Standard, Software Engineering. Product Quality – Part 1: Quality Model, 2001-2004.

[90] C. Izurieta, J. Bieman, The evolution of FreeBSD and linux, ACM/IEEE International Symposium on Empirical Software Engineering (ISESE), 2006.

[91] A. Jansen, P. Avgeriou, J. S. Van der Ven, Enriching software architecture documentation, Journal of Systems and Software, vol. 82, pp. 1232-1248, 2009.

[92] A. Jansen, J. Bosch, Evaluation of tool support for architectural evolution, International Conference on Automated Software Engineering, 2004.

[93] A. G. J. Jansen, Architectural design decisions, ISBN 978-90-367-3494-3, 2008.

[94] M. Jiang, A. Willey, Architecting systems with components and services, International Conference on Information Reuse and Integration (IRI), 2005.

[95] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Carnegie-Mellon University, Software Engineering Institute, 1990.

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

[97] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, The architecture tradeoff analysis method, International Conference on Engineering Complex Computer Systems (ICECCS), 1998.

[98] R. Kazman, S. G. Woods, S. J. Carriere, Requirements for integrating software architecture and reengineering models: CORUM II, Working Conference on Reverse Engineering, 1998.

[99] D. K. Kim, R. France, S. Ghosh, E. Song, A role-based metamodeling approach to specifying design patterns, Annual International Computer Software and Application Conference, 2003.

[100] B. Kitchenham, Procedures for performing systematic reviews, Keele University, TR/SE-0401/NICTA Technical Report 0400011T, vol. 1, 2004.

[101] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering, ISBN 1595933751, Engineering, vol. 2, EBSE 2007-001, 2007.

[102] S. Koch, Evolution of open source software systems–a large-scale investigation, Journal of Software Maintenance and Evolution: Research and Practice, vol. 19, issue 6, pp. 361-382, 2007.

[103] R. Kolb, D. Muthig, T. Patzke, K. Yamauchi, A case study in refactoring a legacy component for reuse in a product line, IEEE International Conference on Software Maintenance (ICSM), 2005.

[104] T. Koponen, Evaluation framework for open source software maintenance, International Conference on Software Engineering Advances (ICSEA), 2006.

[105] G. Kotonya, J. Hutchinson, B. Bloin, A method for formulating and architecting component and service-oriented systems, Book Chapter in Service-Oriented Software System Engineering: Challenges and Practices, ISBN-10 1-59140-426-6, pp. 155-181, 2005.

[106] P. Kruchten, P. Lago, H. van Vliet, Building up and reasoning about architectural knowledge, Quality of Software Architectures, Lecture Notes in Computer Science, vol. 4214, pp. 43-58, 2006.

[107] U. Kulesza, Quantifying the effects of aspect-oriented programming: a maintenance study, IEEE International Conference on Software Maintenance (ICSM), 2006.

[108] U. Kulesza, A. Garcia, C. Lucena, Generating aspect-oriented agent architectures, 3rd Workshop on Early Aspect: Aspect-Oriented Requirements Engineering and Architecture Design, 2004.

[109] P. Lago, P. Avgeriou, R. Capilla, P. Kruchten, Wishes and boundaries for a software architecture knowledge community, Working IEEE/IFIP Conference on Software Architecture (WICSA), 2008.

[110] N. Lassing, D. Rijsenbrij, H. van Vliet, How well can we predict changes at architecture design time?, Journal of Systems and Software, vol. 65, pp. 141-153, 2003.

[111] M. M. Lehman, On understanding laws, evolution, and conservation in the large-program life cycle, Journal of Systems and Software, vol. 1, pp. 213-21, 1980.

[112] M. M. Lehman, D. E. Perry, J. F. Ramil, On evidence supporting the FEAST hypothesis and the laws of software evolution, International Symposium on Software Metrics (METRICS), 1998.

[113] M. M. Lehman, J. F. Ramil, G. Kahen, Evolution as a noun and evolution as a verb, Workshop on Software and Organization Co-evolution (SOCE), 2000.

[114] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, W. M. Turski, Metrics and laws of software evolution - the nineties view, International Software Metrics Symposium, 1997.

[115] X. Liu, B. Iyer, Design architecture, developer networks, and performance of Open Source Software projects, International Conference on Information Systems (ICIS), 2007.

[116] C. H. Lung, S. Bot, K. Kalaichelvan, R. Kazman, An approach to software architecture analysis for evolution and reusability, Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), 1997.

[117] A. Maccari, Experiences in assessing product family software architecture for evolution, International Conference on Software Engineering (ICSE), 2002.

[118] A. Maccari, C. Riva, Architectural evolution of legacy product families, Software Product-Family Engineering, Lecture Notes in Computer Science, vol. 2290, 2002.

[119] N. H. Madhavji, J. Fernandez-Ramil, D. Perry, Software Evolution and Feedback: Theory and Practice, ISBN-10 0470871805, Wiley, 2006.

[120] N. Mansurov, D. Campara, Managed architecture of existing code as a practical transition towards MDA, UML Modeling Languages and Applications, Lecture Notes in Computer Science, vol. 3297, pp. 219-233, 2005.

[121] R. C. Martin, Agile Software Development: Principles, Patterns, and Practices, ISBN 0135974445, Prentice Hall PTR Upper Saddle River, 2003.

[122] M. Matinlassi, Quality-driven software architecture model transformation, Working IEEE/IFIP Conference on Software Architecture (WICSA), 2005.

[123] M. Mattsson, H. Grahn, F. Mårtensson, Software architecture evaluation methods for performance, maintainability, testability, and portability, Conference on the Quality of Software Architecture (QoSA), 2006.

[124] T. J. McCabe, A complexity measure, IEEE Transactions on Software Engineering, pp. 308-320, 1976.

[125] J. A. McCall, P. K. Richards, G. F. Walters, Factors in software quality, National Technical Information Service, 1977.

[126] N. Medvidovic, R. N. Taylor, D. S. Rosenblum, An architecture-based approach to software evolution, International Workshop on the Principles of Software Evolution, 1998.

[127] K. Mens, T. Tourwé, Evolution issues in aspect-oriented programming, Book chapter in Software Evolution (ISBN 978-3-540-76439-7), 2008.
[128] T. Mens, S. Demeyer, Software Evolution, ISBN 978-3-540-76439-7, Springer, 2008.
[129] E. Y. Nakagawa, E. P. M. de Sousa, K. de Brito Murata, G. de Faria Andery, L. B. Morelli, Software architecture relevance in open source software evolution: a case study, International Computer Software and Applications Conference, 2008.
[130] C. L. Nehaniv, P. Wernick, Introduction to software evolvability, International IEEE Workshop on Software Evolvability, 2007.
[131] R. L. Nord, W. G. Wood, P. C. Clements, Integrating the quality attribute workshop (QAW) and the attribute-driven design (ADD) method, Technical Note CMU/SEI-2004-TN-017, 2004.
[132] L. Northrop, P. H. Feiler, B. Pollak, D. Pipitone, Ultra-large-scale systems: the software challenge of the future: Software Engineering Institute, Carnegie Mellon University Pittsburgh, 2006.
[133] L. O'Brien, P. Merson, L. Bass, Quality attributes for service-oriented architectures, International Workshop on Systems Development in SOA Environments (SDSOA), 2007.
[134] W. F. Opdyke, Refactoring object-oriented frameworks, University of Illinois, 1992.
[135] M. Ortega, M. Pérez, T. Rojas, Construction of a systemic quality model for evaluating a software product, Journal of Software Quality, vol. 11, pp. 219-242, 2003.
[136] R. E. Park, Software size measurement: a framework for counting source statements, CMU/SEI-92-TR-20, Software Engineering Institute, Carnegie Mellon University, 1992.
[137] D. L. Parnas, Software aging, International Conference on Software Engineering (ICSE), 1994.
[138] J. W. Paulson, G. Succi, A. Eberlein, An empirical study of open-source and closed-source software products, IEEE Transactions on Software Engineering, vol. 30, pp. 246-256, 2004.
[139] D. E. Perry, A. L. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes, vol. 17, pp. 40-52, 1992.
[140] N. Pessemier, L. Seinturier, T. Coupaye, L. Duchien, A model for developing component-based and aspect-oriented systems, Software Composition, Lecture Notes in Computer Science, vol. 4089, pp. 259-274, 2006.

[141] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering: Foundations, Principles, and Techniques, ISBN-10 3-540-24372-0, Springer, 2005.

[142] C. Raistrick, Applying MDA and UML in the development of a healthcare system, UML Modeling Languages and Applications, Lecture Notes in Computer Science, vol. 3297, pp. 203-218, 2005.

[143] U. Raja, D. P. Hale, J. E. Hale, Modeling software evolution defects: a time series approach, Journal of Software Maintenance and Evolution: Research and Practice, vol. 21, pp. 49-71, 2008.

[144] A. Rashid, A. Garcia, A. Moreira, Aspect-oriented software development beyond programming, The 28th International Conference on Software Engineering (ICSE), 2006.

[145] A. Rashid, A. Moreira, J. Araújo, Modularisation and composition of aspectual requirements, The 2nd International Conference on Aspect-Oriented Software Development (AOSD), 2003.

[146] A. Rawashdeh, B. Matalkah, A new software quality model for evaluating COTS components, Journal of Computer Science, vol. 2, pp. 373-381, 2006.

[147] S. T. Redwine Jr, W. E. Riddle, Software technology maturation, International Conference on Software Engineering (ICSE), 1985.

[148] T. Reus, H. Geers, A. van Deursen, Harvesting software systems for MDA-based reengineering, Model Driven Architecture - Foundations and Applications, Lecture Notes in Computer Science, vol. 4066, pp. 213-225, 2006.

[149] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, I. Herraiz, Evolution and growth in large libre software projects, International Workshop on Principles of Software Evolution (IWPSE), 2005.

[150] G. Robles, J. M. González-Barahona, D. Izquierdo-Cortazar, I. Herraiz, Tools for the study of the usual data sources found in libre software projects, International Journal of Open Source Software and Processes, vol. 1, pp. 24–45, 2009.

[151] G. Robles, J. M. Gonzalez-Barahona, J. J. Merelo, Beyond source code: the importance of other artifacts in software development (a case study), Journal of Systems & Software, vol. 79, pp. 1233-1248, 2006.

[152] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, J. J. Amor, Mining large software compilations over time: another perspective of software evolution, International Workshop on Mining Software Repositories (MSR), 2006.

[153] D. Rowe, J. Leaney, Evaluating evolvability of computer based systems architectures-an ontological approach, Workshop on Engineering of Computer-Based Systems (ECBS), 1997.

[154] D. Rowe, J. Leaney, D. Lowe, Defining systems evolvability-a taxonomy of change, International Conference and Workshop: Engineering of Computer-Based Systems (ECBS), 1998.

[155] W. W. Royce, Managing the development of large software systems: concepts and techniques, International Conference on Software Engineering (ICSE), 1987.

[156] T. L. Saaty, The Analytical Hierarchy Process: Planning, Priority Setting, Resource Allocation, ISBN 0-07-054371-2, McGraw-Hill, 1980.

[157] A. Sampaio, R. Chitchyan, A. Rashid, P. Rayson, EA-Miner: a tool for automating aspect-oriented requirements identification, The 20th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2005.

[158] W. Scacchi, Models of software evolution: life cycle and process, SEI Curriculum Module SEI-CM-10-1.0, 1987.

[159] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, A. J. Offutt, Maintainability of the Linux kernel, IEE Proceedings of Software, vol. 149, pp. 18-23, 2002.

[160] K. Schmid, I. John, R. Kolb, G. Meier, Introducing the PuLSE approach to an embedded system population at Testo AG, International Conference on Software Engineering (ICSE), 2005.

[161] D. Shirtz, M. Kazakov, Y. Shaham-Gafni, Adopting model driven development in a large financial organization, 3rd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), 2007.

[162] M. M. Simmons, P. Vercellone-Smith, P. A. Laplante, Understanding open source software through software archaeology: the case of Nethack, Annual IEEE/NASA Software Engineering Workshop (SEW), 2006.

[163] D. Smith, L. O'Brien, J. Bergey, Using the options analysis for reengineering (OAR) method for mining components for a product line, Software Product Lines, Lecture Notes in Computer Science, vol 2379, 2002.

[164] N. Smith, A. Capiluppi, J. F. Ramil, A study of open source software evolution data using qualitative simulation, Software Process Improvement and Practice, vol. 10, pp. 287-300, 2005.

[165] M. Staron, Adopting model driven software development in industry - a case study at two companies, Model Driven Engineering

Languages and Systems, Lecture Notes in Computer Science, vol. 4199, pp. 57-72, 2006.

[166] C. Stoermer, L. O'Brien, MAP - mining architectures for product line evaluations, Working IEEE/IFIP Conference on Software Architecture (WICSA), 2001.

[167] S. D. Suh, I. Neamtiu, Studying software evolution for taming software complexity, 21st Australian Software Engineering Conference, 2009.

[168] K. J. Sullivan, P. Chalasani, S. Jha, V. Sazawal, Software design as an investment activity: a real options perspective, Real Options and Business Strategy: Applications to Decision Making, pp. 215–262, 1999.

[169] L. G. Thomas, S. R. Schach, G. Z. Heller, J. Offutt, Impact of release intervals on empirical research into software evolution, with application to the maintainability of Linux, Software, IET, vol. 3, pp. 58-66, 2009.

[170] S. A. Tonu, A. Ashkan, L. Tahvildari, Evaluating architectural stability using a metric-based approach, Conference on Software Maintenance and Reengineering (CSMR), 2006.

[171] J. B. Tran, M. W. Godfrey, E. H. S. Lee, R. C. Holt, Architectural repair of open source software, International Workshop on Program Comprehension (IWPC), 2000.

[172] F. van der Linden, J. Bosch, E. Kamsties, K. Kansala, H. Obbink, Software product family evaluation, Software Product Lines, Lecture Notes in Computer Science, vol. 3154, 2004.

[173] A. van Deursen, The software evolution paradox: an aspect mining perspective, International ERCIM Workshop on Software Evolution, 2006.

[174] J. van Gurp, J. Bosch, Design erosion: problems and causes, Journal of Systems & Software, vol. 61, pp. 105-119, 2002.

[175] G. Wang, C. K. Fung, Architecture paradigms and their influences and impacts on component-based software systems, 37th Annual Hawaii International Conference on System Sciences (HICSS), 2004.

[176] Y. Wang, D. Guo, H. Shi, Measuring the evolution of open source software systems with their communities, ACM SIGSOFT Software Engineering Notes, vol. 32, p. 7, 2007.

[177] N. H. Weiderman, J. K. Bergey, D. B. Smith, S. R. Tilley, Approaches to legacy system evolution, CMU/SEI-97-TR-014, Carneige Mellon University, Software Engineering Institute, 1997.

[178] T. Weigert, F. Weil, K. Marth, P. Baker, C. Jervis, P. Dietz, Y. Gui, A. Van Den Berg, K. Fleer, D. Nelson, Experiences in deploying model-driven engineering, SDL 2007: Design for Dependable Systems, Lecture Notes in Computer Science, vol. 4745, pp. 35-53, 2007.

[179] C. Wohlin, M. Höst, P. Runeson, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering: an introduction, ISBN 0-7923-8682-5, Kluwer Academic Pub, 2000.

[180] G. Xie, J. Chen, I. Neamtiu, Towards a better understanding of software evolution: an empirical study on open source software, International Conference on Software Maintenance, 2009.

[181] H. Yang, M. Ward, Successful Evolution of Software Systems, ISBN 1-58053-349-3, Artech House, 2003.

[182] S. S. Yau, J. S. Collofello, T. MacGregor, Ripple effect analysis of software maintenance, International Computer Software and Applications Conference, 1978.

[183] R. K. Yin, Case Study Research: Design and Methods, ISBN-10 0761925538, Sage Publications Inc, 2002.

[184] L. Yu, Indirectly predicting the maintenance effort of open-source software, Journal of Software Maintenance and Evolution: Research and Practice, vol. 18, pp. 311-332, 2006.

[185] L. Yu, S. Ramaswamy, J. Bush, Symbiosis and software evolvability, IT Professional, vol. 10, pp. 56-62, 2008.

[186] M. V. Zelkowitz, D. Wallace, Experimental validation in software engineering, Journal of Information and Software Technology, vol. 39, pp. 735-743, 1997.