

Mälardalen University Press Dissertations

Software Architecture Evolution  
through Evolvability Analysis

Hongyu Pei Breivold

2011

Mälardalen University  
School of Innovation, Design and Engineering



# 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 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

To be written.

Hongyu Pei Breivold  
Shenyang, July, 2011



---

# Contents

<b>Chapter 1. Introduction.....</b>	<b>3</b>
1.1 Research Motivation.....	4
1.2 Research Context.....	4
1.2.1 Proprietary Systems in Focus.....	5
1.2.2 Open Source Software as Complementary Focus.....	5
1.2.3 “How” Perspective of Software Evolution in Focus.....	6
1.2.4 Software Architecture Evolution in Focus.....	6
1.2.5 Architectural Analysis Techniques in Focus.....	7
1.3 Research Questions.....	7
1.4 Research Contributions.....	8
1.4.1 Description of Key Publications.....	10
1.4.2 Other Related Publications.....	17
1.5 Research Methodology.....	19
1.5.1 Research Process.....	19
1.5.2 Research Methods.....	22
1.5.3 Validity.....	24
1.6 Thesis Overview.....	26
<b>Chapter 2. Software Architecture and Evolution.....</b>	<b>29</b>
2.1 Software Architecture.....	29
2.2 Software Evolution.....	31
2.2.1 Laws of Software Evolution.....	31
2.2.2 Properties of Large Software Systems.....	32
2.2.3 Software Aging.....	34
2.3 Software Architecture Evolution.....	35
2.4 Software Quality Models.....	36
2.4.1 McCall’s Quality Model.....	36
2.4.2 Boehm’s Quality Model.....	37
2.4.3 FURPS Quality Model.....	37
2.4.4 ISO 9126 Quality Model.....	38
2.4.5 Dromey’s Quality Model.....	38
2.4.6 Analysis of Software Evolvability in Quality Models.....	39

---

2.5	Software Process Models.....	43
2.6	Techniques and Methods Facilitating Architecture Evolution .....	44
2.6.1	Component-Based and Service-Oriented Engineering.....	45
2.6.2	Software Product Line Methods.....	46
2.6.3	Aspect-Oriented Software Development.....	49
2.6.4	Model-Driven Development.....	51
2.6.5	Reverse Engineering and Reengineering .....	52
2.7	Summary.....	53
<b>Chapter 3.</b>	<b>Architecting for Software Evolvability.....</b>	<b>55</b>
3.1	Systematic Literature Review Process.....	56
3.1.1	Review Protocol .....	56
3.1.2	Inclusion and Exclusion Criteria .....	57
3.1.3	Search Process .....	58
3.1.4	Quality Assessment .....	61
3.1.5	Data Extraction and Synthesis.....	63
3.2	Scope of the Systematic Review.....	63
3.3	Overview of the Primary Studies.....	65
3.3.1	Data Sources.....	65
3.3.2	Citation Status .....	67
3.3.3	Temporal View.....	69
3.3.4	Active Research Communities .....	70
3.3.5	Classification of the Primary Studies .....	71
3.4	Quality Considerations during Software Architecture Design .....	75
3.4.1	Quality Attribute Requirement-Focused .....	75
3.4.2	Quality Attribute Scenario-Focused.....	79
3.4.3	Influencing Factor-Focused.....	80
3.5	Quality Evaluation at Software Architecture Level.....	83
3.5.1	Experience-based.....	84
3.5.2	Scenario-based.....	87
3.5.3	Metric-based.....	91
3.6	Economic Valuation in Determining Level of Uncertainty.....	94
3.7	Architectural Knowledge Management.....	98
3.8	Modeling Techniques .....	102
3.9	Impacts on Research and Practice .....	108
3.9.1	Technology Maturation .....	108
3.9.2	Theoretical Foundation and Formalization .....	110
3.9.3	Combination of Approaches.....	111
3.9.4	Tailoring Approaches for Specific Contexts.....	112
3.10	Summary .....	113

---

<b>Chapter 4. Analyzing Software Evolvability .....</b>	<b>117</b>
4.1 Software Evolvability Model.....	118
4.2 Software Architecture Evolvability Analysis Process .....	120
4.3 Qualitative Evolvability Analysis Method .....	124
4.4 Quantitative Evolvability Analysis Method .....	127
4.4.1 Analytic Hierarchy Process .....	128
4.4.2 Quantitative Evolvability Analysis Method .....	130
4.5 Characterization of the Qualitative and Quantitative Methods ....	135
4.5.1 Application Contexts .....	135
4.5.2 Approaches Used in the Analysis Process .....	136
4.5.3 Analysis Output .....	136
4.5.4 Choosing Between Qualitative and Quantitative .....	138
4.6 Summary.....	140
<b>Chapter 5. Analyzing Proprietary Systems .....</b>	<b>141</b>
5.1 Case Study I. Qualitative Software Evolvability Analysis.....	141
5.1.1 Context of the Case Study .....	141
5.1.2 Evolvability Subcharacteristics from Case Perspective .....	143
5.1.3 Applying the Qualitative Evolvability Analysis Method .....	145
5.1.4 Qualitative Evolvability Analysis: Experiences.....	152
5.1.5 Qualitative Evolvability Analysis: Lessons Learned .....	153
5.2 Case Study II. Quantitative Software Evolvability Analysis.....	154
5.2.1 Context of the Case Study .....	155
5.2.2 Evolvability Subcharacteristics from Case Perspective .....	156
5.2.3 Applying the Quantitative Evolvability Analysis Method ....	158
5.2.4 Quantitative Evolvability Analysis: Experiences.....	165
5.2.5 Quantitative Evolvability Analysis: Lessons Learned .....	166
5.3 Summary.....	166
<b>Chapter 6. Open Source Software Evolution .....</b>	<b>169</b>
6.1 Systematic Literature Review Process.....	170
6.1.1 Review Protocol .....	170
6.1.2 Inclusion and Exclusion Criteria .....	171
6.1.3 Search Process .....	171
6.1.4 Data Extraction and Synthesis.....	172
6.2 Overview of the Primary Studies.....	172
6.2.1 Demographic Information of the Primary Studies .....	173
6.2.2 Categories of the Primary Studies .....	173
6.3 OSS Evolution Trends and Patterns .....	174
6.3.1 Software Growth .....	174

---

6.3.2	Software Maintenance and Evolution Economics.....	177
6.3.3	Prediction of Software Evolution.....	178
6.4	Evolution Process Support.....	179
6.5	Evolvability Characteristics.....	180
6.5.1	Determinism.....	180
6.5.2	Code Understandability.....	180
6.5.3	Complexity.....	181
6.5.4	Modularity.....	182
6.6	Examining OSS at Software Architecture Level.....	183
6.7	Summary.....	184
<b>Chapter 7.</b>	<b>Validity Discussions.....</b>	<b>187</b>
7.1	Validity Aspects on Software Evolvability Model.....	187
7.2	Validity Aspects on AREA Process.....	188
7.3	Validity Aspects on Architecting for Software Evolvability.....	191
7.4	Validity Aspects on Open Source Software Evolution.....	192
<b>Chapter 8.</b>	<b>Conclusions and Future Work.....</b>	<b>195</b>
8.1	Research Questions and Answers.....	195
8.2	Contributions.....	198
8.2.1	Main Research Contributions.....	198
8.2.2	Supplementary Research Contribution.....	200
8.3	Future Research Directions.....	200
<b>Appendix A:</b>	<b>Primary Studies in Chapter 3.....</b>	<b>203</b>
<b>Appendix B:</b>	<b>Primary Studies in Chapter 6.....</b>	<b>211</b>
<b>References</b>	<b>.....</b>	<b>215</b>









## Chapter 1. Introduction

It has long 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 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. In this aspect, 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, language 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. 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.

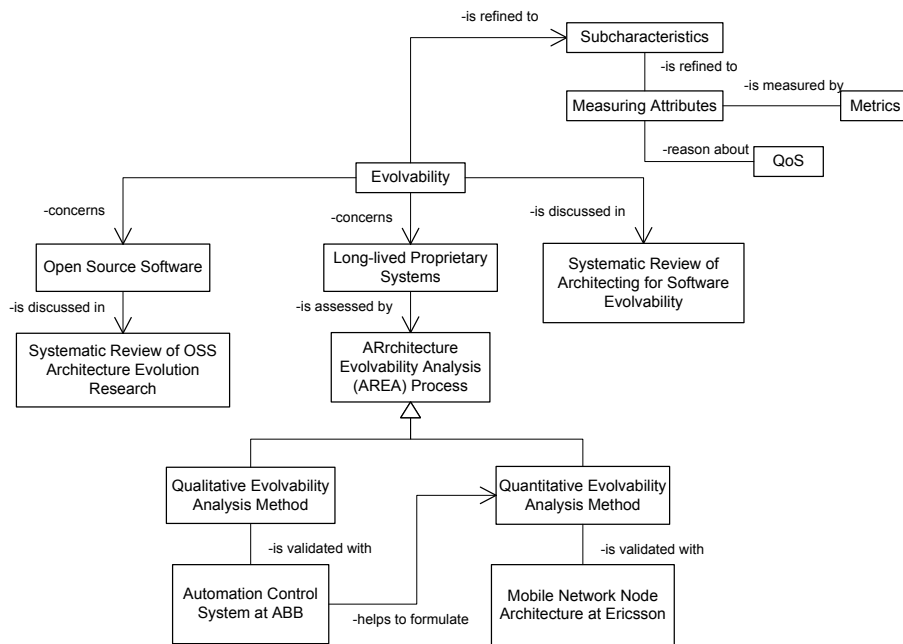
- **Systematic review of architecting for software evolvability**

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 Science of Computer Programming (SCICO) - Special Issue on Software Evolution, Adaptability and Maintenance, 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 in 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.

**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.

**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.

- 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.

**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.

**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.

#### **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 in being able to on forehand understand and analyze systematically the impact of a change stimulus, we introduce a software evolvability model, in which subcharacteristics of software evolvability and corresponding measuring attributes are identified. In addition, a further study of one particular measuring attribute, i.e., modularity, is performed through a dependency analysis case study. We introduce a method for analyzing software evolvability at the architecture level. This is to ensure that the implications of the potential improvement strategies and evolution path of the software architecture are analyzed with respect to the evolvability subcharacteristics. This method is proposed and piloted in an industrial setting. The fact that change stimuli come from both technical and business perspectives spawns two aspects that we also look into in this research, i.e., to respectively investigate the impacts of technology-type and business-type of change stimuli.

**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 Afeef Chauhan, Muhammad Ali Babar, 17<sup>th</sup> 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.

**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.

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

- 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 in 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.

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

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

- Analysis of Software Evolvability in Quality Models
  - Hongyu Pei Breivold, Ivica Crnkovic, 35<sup>th</sup> 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.

**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.

**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.

- Analyzing Software Evolvability
  - Hongyu Pei Breivold, Ivica Crnkovic, Peter Eriksson, 32<sup>nd</sup> 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.

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

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

- Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study
  - Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, Magnus Larsson, 3<sup>rd</sup> 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.

**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.

**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.

## 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, 5<sup>th</sup> 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, 21<sup>st</sup> 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, 21<sup>st</sup> 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, 34<sup>th</sup> 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, 4<sup>th</sup> 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, 33<sup>rd</sup> 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, 31<sup>st</sup> 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

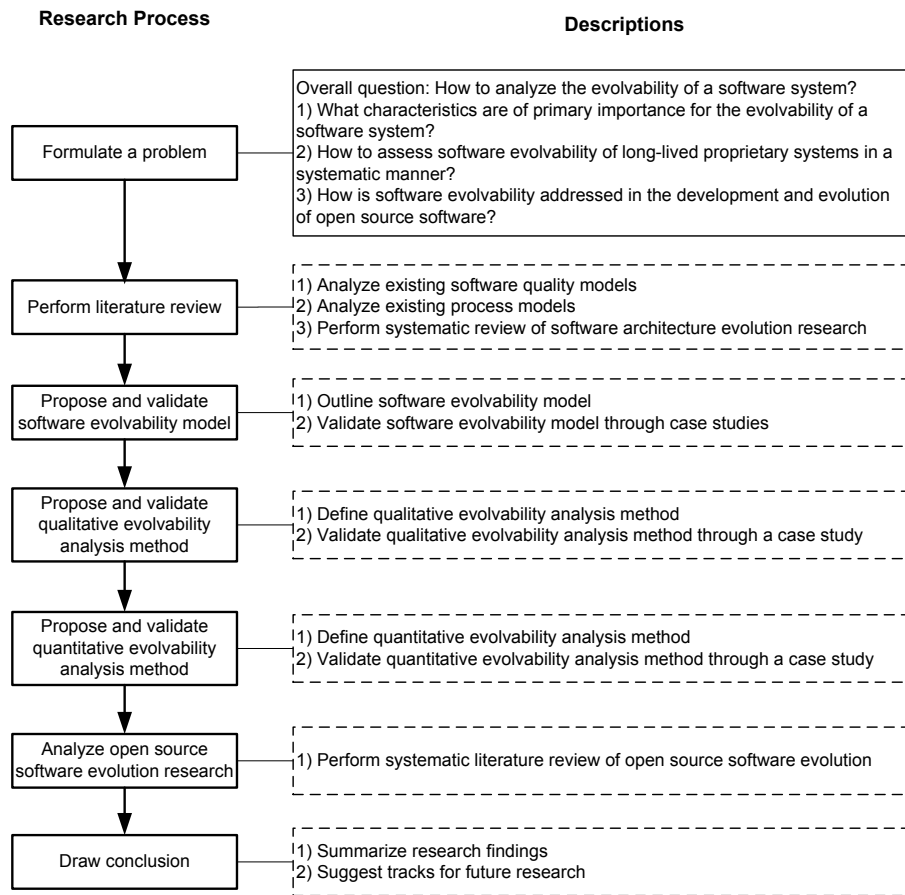
- Using Software Evolvability Model for Evolvability Analysis
  - Hongyu Pei Breivold, Ivica Crnkovic, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-222/2008-1-SE, Mälardalen Real-Time Research Center, Mälardalen University, February, 2008

## 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 2-2: 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. Besides, a characterization of a proprietary system, and a software evolvability model were 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 extension 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, i.e., 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** This chapter 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** This chapter presents relevant fields of research and practice in software architecture and its evolution.

**Chapter 3 – Architecting for Software Evolvability** This chapter 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** This chapter 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** This chapter 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** This chapter 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** This chapter discusses in details the validity aspects of the research results.

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

**Appendix A – Primary Studies in Chapter 3** This appendix 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** This appendix 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

This chapter relates the work in this thesis to relevant research and practice areas, subdivided into a number of sections, including software architecture evolution, software quality models, software process models, as well as techniques that facilitate architecture evolution. In each section, there is also an explanation of how the thesis is related to each area.

### 2.1 Software Architecture

This section provides a brief overview over software architecture, which plays the central role in this thesis to address software evolution challenge.

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*”.

A system inhabits an environment that can influence the system. A system has one or more stakeholders. Each stakeholder has interests in, or relative to, that system. A *system stakeholder* is an individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system [88].

*Concerns* are those 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, security, distribution, and evolvability [88].

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* which outlines a design for the software of a system.

- *Roadmap* which allows to plan 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* which enables different stakeholders to communicate about the major decisions in order to steer and influence the software of a system.
- *Quality predictor* which 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 presented 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.

In this thesis, we focus on software architecture evolution, and therefore, the software architecture purposes with respect to *roadmap*, *communication vehicle*, and *quality predictor* play visible roles in our research.

## 2.2 Software Evolution

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

### 2.2.1 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.

## 2.2.2 Properties of Large Software Systems

The following properties of large software systems are noted by Brooks [38]:

- *Complexity*

This 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.

- *Conformity*

Many software systems are constrained by the need to conform to human institutions and systems.

- *Changeability*

The software entity is constantly subject to pressures for change.

- *Invisibility*

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.

The above-noted properties of large software systems concerning software complexity, inevitable changes of software systems, and invisibility in terms of software structure representation, further confirm the software evolution phenomena described in Chapter 2.2.1. All these exhibit 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, we propose software evolvability model and software architecture evolvability analysis process, and we describe these in Chapter 4 and Chapter 5 of this thesis.

### 2.2.3 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.

## 2.3 Software Architecture 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.

## 2.4 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. 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]. In the following subsections, we provide a brief survey of these 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.

### 2.4.1 McCall's Quality Model

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.

## 2.4.2 Boehm's Quality Model

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.

## 2.4.3 FURPS Quality Model

The characteristics that are taken into consideration in FURPS [83] are:

- *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, installability, 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.

#### 2.4.4 ISO 9126 Quality Model

ISO 9126 [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.

#### 2.4.5 Dromey's Quality Model

Dromey [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 Matakah [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.

#### 2.4.6 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

<b>Extensibility</b>			x		
<b>Flexibility</b>	x				
<b>Human Engineering</b>		x			
<b>Integrity</b>	x				
<b>Interoperability</b>	x			X	
<b>Maintainability</b>	x	x	x	X	x
<b>Modifiability</b>		x		X	
<b>Performance</b>			x		
<b>Portability</b>	x	x		x	x
<b>Reliability</b>	x	x	x	x	x
<b>Reusability</b>	x				x
<b>Supportability</b>			x		
<b>Testability</b>	x	x		x	
<b>Understandability</b>		x		x	
<b>Usability</b>	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
----------------	----------------------------	-------

<b>McCall</b>	The effort required to locate and fix a fault in the program within its operating environment	Corrective maintenance
<b>Boehm</b>	It is concerned with how easy it is to understand, modify and test.	Understandability, modifiability and testability
<b>FURPS</b>	Implicit	Adaptability, extensibility
<b>ISO 9126</b>	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 thesis, 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 for a software system to be evolvable. 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**

<b>Characteristics</b>	<b>Evolvability</b>	<b>Maintainability</b>
<b>Software Change Stimuli</b>	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
<b>Type of Change</b>	Coarse-grained, long term, higher level, radical functional or structural enhancements or adaptations [177]	Fine-grained, short term, localized change [177]
<b>Focus Activity</b>	Cope with changes	Keep the system perform functions
<b>Software</b>	Structural change	Relatively constant

<b>Structure</b>		
<b>Analysis Scenarios</b>	Growth scenarios (change scenarios)	Existing use case scenarios
<b>Development Process</b>	May require corresponding process changes	Relatively constant
<b>Architecture Integrity</b>	Conformance is required	Conformance is preserved

Studying the existing quality models provides us the research idea of establishing a software evolvability model (see Chapter 4). Moreover, based on our working experiences with various industrial software systems in different domains, we have found out particular quality attributes that are essential for evolvability (i.e., evolvability subcharacteristics, which will be detailed in Chapter 4), e.g., analyzability, architectural integrity, portability, testability, changeability and extensibility. Based on these evolvability subcharacteristics, we have classified the set of quality characteristics (see Table 2-1) covered in the well-known quality models against evolvability subcharacteristics, as shown in Table 2-4. Besides, we have followed ISO 9126 standards [89], and checked their quality attributes against our classification. 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.

**Table 2-4: Classification of quality characteristics in quality models**

<b>Classification</b>	<b>Quality Characteristics in Quality Models</b>
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)
--	---------

## 2.5 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.

## 2.6 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.

### 2.6.1 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.

## 2.6.2 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<sup>TM</sup> 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.

### 2.6.3 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.

## 2.6.4 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.

### 2.6.5 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*

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*

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*

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.

## 2.7 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**

<b>Inclusion Criteria</b>
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.
<b>Exclusion Criteria</b>
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]<sup>1</sup>. 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>)

---

<sup>1</sup> 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.

- 
- 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 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. 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<sup>2</sup>, 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.

---

<sup>2</sup> [www.endnote.com](http://www.endnote.com)

### 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 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.