

Feature-Based Methodology for Supporting Architecture Refactoring and Maintenance of Long-Life Software Systems

Dissertation
Zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von Dipl.-Ing. Ilian Pashov
geboren am 23.06.1975 in Svistov, Bulgarien

Tag der Einreichung: 10.02.2004
Tag der wissenschaftlichen Aussprache: 08.10.2004

Gutachter: 1.) Prof. Dr.-Inzg. habil. Ilka Philippow
2.) Prof. Dr. Wilhelm Rossak
3.) Dr. Christoph Pesch

Zusammenfassung

Langlebige Software-Systeme durchlaufen viele bedeutende Veränderungen im Laufe ihres Lebenszyklus, um der Weiterentwicklung der Problemdomänen zu folgen. Normalerweise ist es schwierig eine Software-Systemarchitektur den schnellen Weiterentwicklungen einer Problemdomäne anzupassen und mit der Zeit wird der Unterschied zwischen der Problemdomäne und der Software-Systemarchitektur zu groß, um weitere Softwareentwicklung sinnvoll fortzuführen. Fristgerechte Refactorings der Systemarchitektur sind notwendig, um dieses Problem zu vermeiden.

Aufgrund des verhältnismäßig hohen Gefahrenpotenzials und des zeitlich stark verzögerten Nutzens von Refactorings, werden diese Maßnahmen normalerweise bis zum letztmöglichen Zeitpunkt hinausgeschoben. In der Regel ist das Management abgeneigt Architektur-Refactorings zu akzeptieren, außer diese sind absolut notwendig. Die bevorzugte Vorgehensweise ist, neue Systemmerkmale ad hoc hinzuzufügen und nach dem Motto "Ändere nie etwas an einem funktionierenden System!" vorzugehen. Letztlich ist das Ergebnis ein Architekturzerfall (Architekturdrift). Die Notwendigkeit kleiner Refactoring-Schritte führt zur Notwendigkeit des Architektur-Reengineerings. Im Gegensatz zum Refactoring, das eine normale Entwicklungstätigkeit darstellt, ist Reengineering eine Form der Software-"Revolution". Reengineeringprojekte sind sehr riskant und kostspielig. Der Nutzen des Reengineerings ist normalerweise nicht so hoch wie erwartet. Wenn nach dem Reengineering schließlich die erforderlichen Architekturänderungen stattfinden, kann dies zu spät sein. Trotz der enormen in das Projekt gesteckten Bemühungen erfüllen die Resultate des Reengineerings normalerweise nicht die Erwartungen. Es kann passieren, dass sehr bald ein neues, kostspieliges Reengineering erforderlich wird.

In dieser Arbeit werden das Problem der Softwareevolution und der Zerfall von Softwarearchitekturen behandelt. Eine Methode wird vorgestellt, welche die Softwareentwicklung in ihrer entscheidenden Phase, dem Architekturrefactoring, unterstützt. Die Softwareentwicklung wird sowohl in technischer als auch organisatorischer Hinsicht unterstützt. Diese Arbeit hat neue Techniken entwickelt, welche die Reverse-Engineering-, Architecture-Recovery- und Architecture-Redesign-Tätigkeiten unterstützen. Sie schlägt auch Änderungen des Softwareentwicklungsprozesses vor, die fristgerechte Architekturrefactorings erzwingen können und damit die Notwendigkeit der Durchführung eines Architektur-Reengineerings vermeiden.

In dieser Arbeit wird die Merkmalmodellierung als Hauptinstrument verwendet. Merkmale werden genutzt, um die Abstraktionslücke zwischen den Anforderungen der Problemdomäne und der Systemarchitektur zu füllen. Merkmalmodelle werden auch als erster Grundriss für die Wiederherstellung der verlorenen Systemarchitektur genutzt. Merkmalbasierte Analysen führen zu diversen, nützlichen Hinweisen für den erneuten Entwurf (das Re-Design) einer Architektur. Schließlich wird die Merkmalmodellierung als Kommunikationsmittel zwischen unterschiedlichen Projektbeteiligten (Stakeholdern) im Verlauf des Softwareengineering-Prozesses verwendet und auf dieser Grundlage wird ein neuer Anforderungsdefinitionsprozess vorgeschlagen, der die erforderlichen Architekturrefactorings erzwingt.

Abstract

The long-life software systems withstand many significant changes throughout their life-cycle in order to follow the evolution of the problem domains. Usually, the software system architecture can not follow the rapid evolution of a problem domain and with time, the diversion of the architecture in respect to the domain features becomes prohibiting for software evolution. For avoiding this problem, periodical refactorings of the system architecture are required.

Usually, architecture refactorings are postponed until the very last moment, because of the relatively high risk involved and the lack of short-term profit. As a rule, the management is unwilling to accept architecture refactorings unless they become absolutely necessary. The preferred way of working is to add new system features in an ad-hoc manner and to keep the rule "Never touch a running system!". The final result is an architecture decay. The need of performing small refactoring activities turns into need for architecture reengineering. In contrast to refactoring, which is a normal evolutionary activity, reengineering is a kind of software "revolution". Reengineering projects are risky and expensive. The effectiveness of reengineering is also usually not as high as expected. When finally after reengineering the required architecture changes take place, it can be too late. Despite the enormous invested efforts, the results of the reengineering usually do not satisfy the expectations. It might happen that very soon a new expensive reengineering is required.

This thesis deals with the problem of software evolution and the decay of software architectures. It presents a method, which assists software evolution in its crucial part, the architecture refactoring. The assistance is performed for both technical and organizational aspects of the software evolution. The thesis provides new techniques for supporting reverse engineering, architecture recovery and re-designing activities. It also proposes changes to the software engineering process, which can force timely architecture refactorings and thus avoid the need of performing architecture reengineering.

For the work in this thesis feature modeling is utilized as a main asset. Features are used to fill the abstraction gap between domain requirements and system architecture. Feature models are also used as an outline for recovering of lost system architectures. Through feature-based analyses a number of useful hints and clues for architecture redesign are produced. Finally, feature modeling is used as a communication between different stakeholders of the software engineering process and on this basis a new requirements engineering process is proposed, which forces the needed architecture refactorings.

Acknowledgements

First I have to thank my parents for giving me the best of their life and supporting my education with all means, even though there have been some very difficult moments for them, moments requiring many compromises and sacrifices. Thanks Mum, Thanks Dad!

I have to appreciate the irreplaceable support of my girl friend Dilia, who has given me awareness of my skills and has provided me with an invaluable support for my education, my work and my personal development. Thanks Doly!

I would like to thank my supervisor Prof. Philippow and also to Dr. Riebisch from the Technical University of Ilmenau for providing the constructive and pleasant working atmosphere at the university, for guiding my research and giving me precious and wise advices. Thank You Prof. Philippow, thanks Matthias and I wish both of you many successful years at the difficult academic field. I wish you would continue to be always so dedicated and open to the young people willing to develop themselves.

This work was held in the context of a project between the Technical University of Ilmenau and the Postal Automation Division of Siemens Dematic AG in Constance, Germany . The work would not have been possible without the support of the Siemens Dematic system architects and especially without the irreplaceable help of their head man Mr. Michael Zettler. Thank you Michael, you are the proof that the so necessary openness of the industry to the academic world exists. Thank you for giving me your support over all these years.

I would also like to give my special regards to all my colleagues and friends for their constructive critique, which has helped my ideas to elaborate and to become more valuable.

Special thanks to my colleague Periklis Sochos for his assistance concerning the quality of the English language used in this thesis.

Contents

I	The Software Evolution Problem	1
1	What Is This Thesis About?	5
2	Evolution of Software Systems - What Is It?	7
2.1	Evolution Etymology	7
2.2	Life Cycle of Long-Life Software Systems	7
2.3	Evolution as a Part of the Long-Life Software Systems Life Cycle	8
2.4	Definition of Software Evolution	8
2.5	Software Evolution in the Context of Software Maintenance	9
2.6	Software Evolution Related Techniques	10
2.6.1	Program Transformation	10
2.6.2	Reverse Engineering	10
2.6.3	Refactoring	11
2.6.4	Reengineering	11
2.6.5	New Development (Replacement)	11
3	Methodology for Supporting Software Evolution	13
3.1	Motivation	13
3.1.1	Preserving the Need of Software Revolutions or Increasing the Period Between Two of Them	14
3.1.2	Increasing The Overall Life-Time	14
3.1.3	Reducing The Maintenance Costs	14
3.2	Solution	15
3.2.1	Collecting, Structuring and Saving Domain Knowledge Embedded in Software Systems	15
3.2.2	Improving the Effectiveness of Changes	15
3.2.3	Adapting the Software Engineering Process for Supporting Evolution	16
II	State of The Art	19
4	Software Life Cycle Models - Frameworks for Software Evolution	21
4.1	Motivation	21
4.2	Classic Waterfall Model	21
4.3	Incremental Model	21
4.4	Spiral Model	22
4.5	Staged Model	22
4.6	Agile Models for Software Life Cycle	24
4.7	Conclusion	25
5	Domain Engineering - Collecting and Implementing Knowledge for Reuse	27
5.1	Motivation	27
5.2	In a Nutshell	28
5.3	Domain Engineering and Software Evolution	28
5.4	Domain Analysis	29
5.5	Domain Design	30

5.6	Domain Implementation	31
5.7	Domain Engineering Concepts	31
5.7.1	Features and Feature Modeling in Feature Oriented Domain Analysis (FODA)	31
5.7.2	Modeling the system dynamics in Reuse-Driven Software Engineering Business (RSEB)	33
5.7.3	The ”+1” Model of Domain Engineering in the FeatuRSEB Approach	34
5.8	Conclusion	35
6	Feature Modeling	37
6.1	Motivation	37
6.2	Definition	37
6.3	Overview of Feature Models and Feature Modeling	38
6.3.1	Feature Modeling Symbology	38
6.3.2	Grouping of Features and Feature Hierarchy	39
6.3.3	Additional Relations Between Features	40
6.4	Conclusion	41
7	Re-Engineering - Analyzing, Understanding and Reworking Software Systems	43
7.1	Motivation	43
7.2	Program Understanding	43
7.3	Reverse Engineering	45
7.4	Architecture and Design Recovery	45
7.5	Program Transformation	48
7.6	Redundancies and Clones Detection	48
7.7	Refactoring	49
7.8	Conclusion	50
8	Software Architecture - The Design Base for Evolution of Complex Systems	51
8.1	Motivation	51
8.2	Software Architecture Basics	52
8.3	Software Architecture Modeling	53
8.3.1	The 4+1 View Model	53
8.3.2	The 4 View Model	54
8.3.3	Other architecture modeling approaches	55
8.4	Conclusion	56
9	Software Product Lines - A Solution for Evolution of Long-Life Software Systems	57
9.1	Motivation	57
9.2	Software Product Lines Overview	57
9.3	Requirements Engineering for Software Product Lines - Driving the Software Evolution	58
9.4	Conclusion	60
III	New approach	63
10	Extending the Role of Feature Modeling in Software Engineering Process	65
10.1	Motivation	65
10.2	Evolution Requirements for Using Domain Knowledge	65
10.3	Structuring Domain Knowledge with Feature Models	66
10.3.1	Retrieving Design Objectives by Analyzing Customer Needs	66
10.3.2	Retrieving Design Decisions by Analyzing System Solutions	66
10.3.3	Making Overall System Presentation With Feature Models	67
10.4	Bridging the Gap Between Requirements and Architecture	68
10.5	Combining System Dynamics and Feature Models	69
10.6	Overall Structure of an Evolution Friendly Feature Model	70
10.7	Conclusion	71

11 Feature-Assisted Reverse Engineering	73
11.1 Motivation	73
11.2 Establishment of Feature Model for an Existing System	73
11.2.1 Collection of Information Sources	74
11.2.2 Model Design Objectives	75
11.2.3 Model Design Decisions	75
11.2.4 Collect Use Cases	75
11.2.5 Verifying the Feature Model	75
11.3 Using Feature Models in Program Comprehension	76
11.4 Feature-Based Architecture Recovery	76
11.4.1 The Architecture Recovery Process	77
11.4.2 Verifying Architecture Hypotheses With Feature-Architecture Elements Cross-References	79
11.4.3 Tracing Features to Source Code	80
11.4.4 Using an Architecture Model as a Frame for Architecture Hypotheses	80
11.5 Conclusion	81
12 Feature-Assisted Architecture Redesign	83
12.1 Motivation	83
12.2 Feature-Assisted Architecture Redesign - Process and Dataflow	83
12.3 Detection of Architectural Disproportions	85
12.4 Clustering of Features	87
12.5 Detection of Redundancies	91
12.6 Hinting Separation of System Concerns	92
12.7 Deduction of Clues and Hints for Architecture Development	93
12.8 Assuring Completeness and Appropriateness of Architecture Redesign	94
12.9 Conclusion	94
13 Feature-Based Requirements Engineering for Supporting Software Evolution	97
13.1 Motivation	97
13.2 Prerequisites for Performing Feature-Based Requirements Engineering	98
13.3 Feature-Based Requirements Engineering Rules	99
13.4 Feature-Based Requirements Engineering Process	101
13.4.1 Feature-Based Requirements Engineering Actors	101
13.4.2 Feature-Based Requirements Engineering Activities	101
13.4.3 Feature-Based Requirements Engineering Dataflow	104
13.5 Software Evolution by Performing Feature-Based Requirements Engineering	106
13.5.1 The Changed Evolution Process	106
13.5.2 Architecture Versioning Strategy	107
13.5.3 Integrated Architecture Refactoring (continuous migration)	107
13.6 Conclusion	108
IV Proof of Concept	111
14 Applying Feature-Based Techniques within a Postal Automation System - Case Study	113
14.1 Motivation	113
14.2 Restrictions	113
14.3 Introduction	113
14.3.1 The Industrial Environment	113
14.3.2 Challenges to the Feature-Based Methodology	115
14.4 Feature-Assisted Architecture Reengineering of an Industrial Software System	115
14.5 Performing Feature-Based Requirements Engineering in an Industrial Environment	116
14.6 Tool Support	116
14.7 Conclusion	116

V Conclusion	119
15 Summary	121
16 Future Work	123

List of Figures

2.1	Life Cycle of Long-Life Software Systems	8
2.2	Evolution Stages within Long-Life Software Systems	9
2.3	Software Evolution in the Context of Software Maintenance	10
3.1	Increasing The Period Between Two Revolution Changes	14
3.2	Changes of Problem Domain in Comparison to Changes of The System Architecture . .	16
4.1	Waterfall Life Cycle Model	22
4.2	Incremental Life Cycle Model (adapted from [Sommerville 2001])	22
4.3	Spiral Model [Pressman 1991]	23
4.4	The Staged Model for Software Life Cycle [Rajlich et al. 2000]	23
4.5	XP Map [XP]	25
5.1	Vertical and Horizontal Domains	27
5.2	Software development based on Domain Engineering [SEIDE]	29
5.3	Definition of Domain Analysis Process [SEIDE]	29
5.4	Definition of Domain Design Process [SEIDE]	30
5.5	Definition of Domain Implementation Process [SEIDE]	31
5.6	Features vs. Use Cases [Griss et al. 1998]	34
6.1	The Classical "Car Example" of a Feature Model	38
6.2	Used Feature Modeling Symbolology	39
6.3	Feature Model with Multiplicities	40
7.1	Reengineering - adapted from [Nelson 1996]	44
7.2	Reverse engineering - adapted from [Nelson 1996]	45
7.3	Design Recovery - adapted from [Nelson 1996]	46
8.1	Architecture views in "4+1 View" model - adapted from [Kruchten 1995]	53
8.2	Architecture views in "4 View" model - adapted from [Hofmeister et al. 2000]	54
9.1	Iterative Activities of the Development Process for Software Product Lines	58
9.2	Historical Development of Methods for Software Product Lines [Boellert 2002]	59
10.1	Design Objectives in a Studied Image Processing System	67
10.2	Design Decisions in a Studied Image Processing System	67
10.3	Current Relations Between Requirements and Architecture	68
10.4	Breaking the Direct Relation Between Requirements and Architecture	68
10.5	Common Relations Between Requirements, Features and Architecture	69
10.6	Restricted Relations Between Requirements, Features and Architecture	69
10.7	Relating Features and Use Cases	70
10.8	Example of a Feature Model with Layered Structure	70
11.1	Establishing Feature Model of an Existing System	74
11.2	Feature-Assisted Architecture Recovery Process	77
11.3	Image Storage Capabilities - Extraction from Studied Image Processing System Feature Model	78

11.4 Tracing Features to Architecture Elements	79
12.1 Feature-Assisted Architecture Design - Activities and Data Flow	84
12.2 Sample Traceability Links between Features and Architecture Components	85
12.3 Features Clustering Process Activities and Dataflow	88
12.4 Features Clustering Tool - Feature Groups Edit Form	89
12.5 Features Clustering Tool - Feature Groups Assignment Form	89
12.6 Features Clustering Tool - Groups-Features Report (Established Feature Clusters) . . .	90
12.7 Possible Redundancies Detected in Studied Image Processing System	91
12.8 Features Clustering Tool - Similarities Examination Form	92
12.9 Hinting Separation of Concerns - Image Component Example	93
12.10 Newly Established "Image" Component in Comparison to Components from the Old System Architecture of the Studied Image Processing System	93
13.1 Usual Software Evolution Process by Applying Product Line Development	98
13.2 Common Architecture Versioning Strategy	99
13.3 Feature-Based Requirements Engineering Activities	103
13.4 Feature-Based Requirements Engineering Dataflow - Top View	104
13.5 Feature-Based Requirements Engineering Dataflow - Detailed View	105
13.6 Software Evolution Process by Performing Feature-Based Requirements Engineering . .	106
13.7 Architecture Versioning Strategy by Using Feature-Based Requirements Engineering . .	107
13.8 Integrated Architecture Refactoring by Using Feature-Based Requirements Engineering	108
14.1 Typical Architecture of a Reading-Coding System	114

List of Tables

11.1 Features to Architecture Elements Cross-Reference Table	79
11.2 Features to Source Code Cross-Reference Table	80
12.1 Features per component classification example	86
12.2 Words Frequency Analysis Example	87
12.3 Words Categorization Analysis Example - Categorization Tree	88
12.4 Sample Feature Groups and The Corresponding Keywords	90
12.5 Features Classified to "Image TIFF Operations" and "Image IO Operations" Groups . .	90
12.6 Similar Features From The "Image TIFF Operations" Group	92

Part I

The Software Evolution Problem

"If you have a problem, solve it in motion."

Diliana Dancheva Simeonova, 1996

Chapter 1

What Is This Thesis About?

This thesis examines the issue of outdated the system architecture by software products, which have a long life cycle. The thesis presents a methodology that assists refactoring and maintenance of software architectures. Inevitably, both take place in the life cycle of the long-life software systems and are parts of the software evolution. The presented method utilizes the assets of features and feature modeling. It shows how feature models can assist reverse engineering and later restructuring of software architectures. On the basis of feature models new requirements engineering activities are introduced, which support the architecture maintenance. The methodology states the fundamentals of making software evolution more methodologically.

The work in the thesis is divided in five parts. Parts one and two give a comprehensive overview of software evolution and the related technology. Part three presents a new approach for supporting the software evolution. Part four proves the concepts presented in part three and shows how the developed approach was applied in an industrial project. Part five summarizes the work and outlines the open topics for future research.

Chapter 2

Evolution of Software Systems - What Is It?

Software systems are usually designed to provide a solution to a specific problem domain and for a specific business case. Since the business world is constantly changing and the problem domains evolve the software systems have to be continuously adapted to new business needs, i.e. they need to evolve. This is particularly true for software that has served a problem domain over many years. Taking a look at the life cycle of such systems one can see that evolution has become an inseparable part of it.

Over the last decade, software evolution has increasingly become a subject of discussion and research. More and more areas of software engineering are being related to it. People start to realize its importance. Still, software evolution lacks a common understanding and standard definition. That is why it is necessary to define the context and the way in which software evolution should be understood in this thesis.

2.1 Evolution Etymology

The word *evolution* dates back to 1662. It originates from the Latin verb *evolvere*, which means unroll or disclose. In contemporary language, the word evolution has a meaning of a process of continuous change from a lower, simpler, or worse to a higher, more complex, or better state. Additionally, evolution means the process of working out or developing.

If the changes required for an evolution are made suddenly, radically and alter the foundations of the evolution subject, then one is talking about *revolution*.

The term evolution is widely used in different science areas and has lately been applied in software engineering as well. In the next sections the term is explained and defined with its exact meaning for the way it should be understood from now on in this thesis.

2.2 Life Cycle of Long-Life Software Systems

In order to establish a clear meaning of software evolution, one needs a short introduction into the life cycle of software systems. Since evolution is a term, which carries the meaning of continuance in itself, one should expect to find it especially by systems with long life time expectancy. Usually these are the systems, which serve in a problem domain over decades. They provide solutions for business cases, which can be considered as more conservative. Examples are the information systems for banking or insurance companies, postal automation systems and etc.

The software systems as a product of the software industry have their own life cycle. It differs depending on the business case and respectively on life time expectancy of the systems. The software products with short life time expectancy are usually designed to solve a particular business case, which is clearly identified and not expected to be a subject to change. Respectively the product life cycle is similar to the standard one, which is usually described in the economics theory with six phases.

- Development
- Introduction

- Growth
- Maturity
- Decline

The most important characteristic for such software systems is that changes are rarely applied on them after the development phase. If changes are applied, they can be considered minor and do not concern major system attributes like software architecture.

On the contrary - software products with long life time expectancy are more often subject to changes, many of which concern system architecture. Such systems are supposed to support business needs over many years and since the business domain evolves, software systems also need to evolve.

The frequent changes make the standard life cycle model inapplicable for long-life software systems. The development within these systems is not performed within a single step, but separated over multiple ones. Practice showed that incremental models are more suitable for this case. These models do not focus development efforts in one single phase, but they introduce different stages, during which the software is adapted to new business requirements. The presence of such stages is the main difference, distinguishing between the life cycle of long life software and systems with short life expectancy.

As one can see on Figure 2.1, the development of long-life software systems never stops. There are constant development activities, which adapt the systems to the business needs. The results of these activities are new releases, featuring a new state of the system architecture. The last is another special feature of the long-life software systems - their system architecture is subject to change and goes through many states during the system life time.

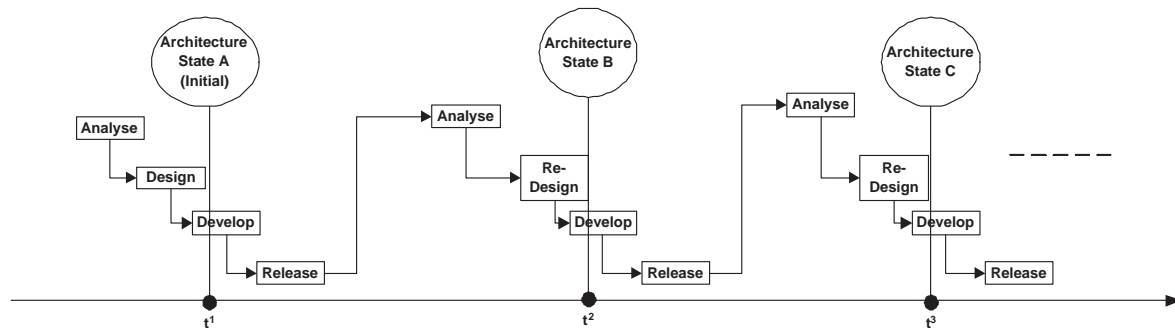


Figure 2.1: Life Cycle of Long-Life Software Systems

2.3 Evolution as a Part of the Long-Life Software Systems Life Cycle

As it was shown in the previous section, the long-life software systems characterize themselves with continuous changes from one state to another. This property makes the term evolution especially applicable for these systems and the evolution itself an inseparable part of their life cycle. If one looks at Figure 2.1 again, the different stages after the initial development where the system is further developed can be considered as a continuous evolution (Figure 2.2).

2.4 Definition of Software Evolution

There is no standard definition for the term software evolution in the vocabulary of software engineering. Some researchers and practitioners use it as a preferable substitute for maintenance. In the "Staged Model for The Software Life Cycle" ([Bennett et al. 2000]) software evolution is defined as *a particular stage in the software life cycle*. It is the stage after the initial development, the stage at which the system is subject to evolution changes. This is rather a recursive definition, but it stays closest to the meaning, which is used in this thesis from now on. The work in the thesis stays close to the context of the staged model, but additionally it concerns aspects of the working definition of software evolution given by Davor Svetinovic [Svetinovic 2003]. According to him *software evolution is*

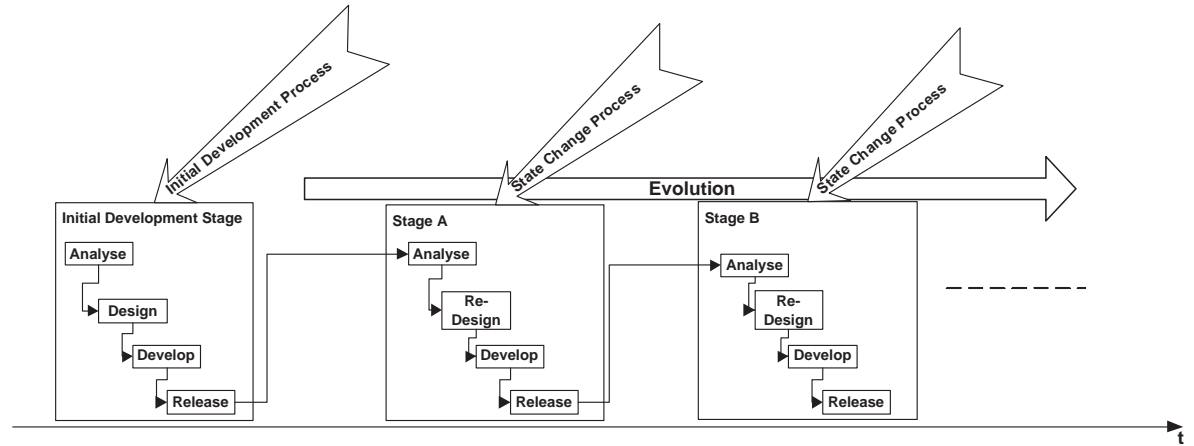


Figure 2.2: *Evolution Stages within Long-Life Software Systems*

change in essential properties (attributes) of a product family over time. Another definition, which has to be taken into account is the one provided from the Research Institute in Software Evolution: *the set of activities, both technical and managerial, that ensures that software continues to meet organizational and business objectives in a cost effective way* [RISE 2003].

Based on the above mentioned definitions and in the context of long-life software systems, software evolution is defined as:

Software evolution is the process of continuous changing essential attributes such as the system architecture of a software system. Software evolution is a repeating process for systems with long lifetime expectancy, which determines a particular stage in the software life cycle. Software evolution assures that software continues to meet organizational and business objectives in a cost effective way.

The above definition outlines the main aspects of the software evolution, which are important in the context of this thesis. First, the software evolution is a repeating process and as such, it requires methodical work. Second, software evolution requires changes of main system attributes and the success of these changes determines the success of the evolution. As it is shown in the next chapter these two aspects of the software evolution determine in a great extent the aims and the frame of the presented work.

2.5 Software Evolution in the Context of Software Maintenance

Often, software evolution is referred to the activities taking place after a software product has been delivered to the customer, i.e. the well-known software maintenance. Software maintenance is defined by the ANSI/IEEE Standard 729-1983 as: *modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.*

Without contradicting the above statement there is a need to precise the way in which evolution should be understood in the context of software maintenance.

It is true that evolution takes place after a software system has progressed throughout its initial development phase and is in its maintenance phase according to the definition software maintenance. But it should be well understood that for the long life software systems the maintenance phase continues much longer than the initial development and during the maintenance the software systems pass through significant changes. Some of these changes require a lot of further development and some of them are only servicing ones. The changes connected to the further development of major system attributes, such as software architecture, determine the software evolution. The evolution can be considered as a sub-process of the big maintenance process, which begins after the initial system delivery. But evolution is comparable to and even more massive than the initial development. It can be the case, when evolution results in a new deliverable version of a software system, requiring its own maintenance. Figure 2.3 illustrates the last statements.

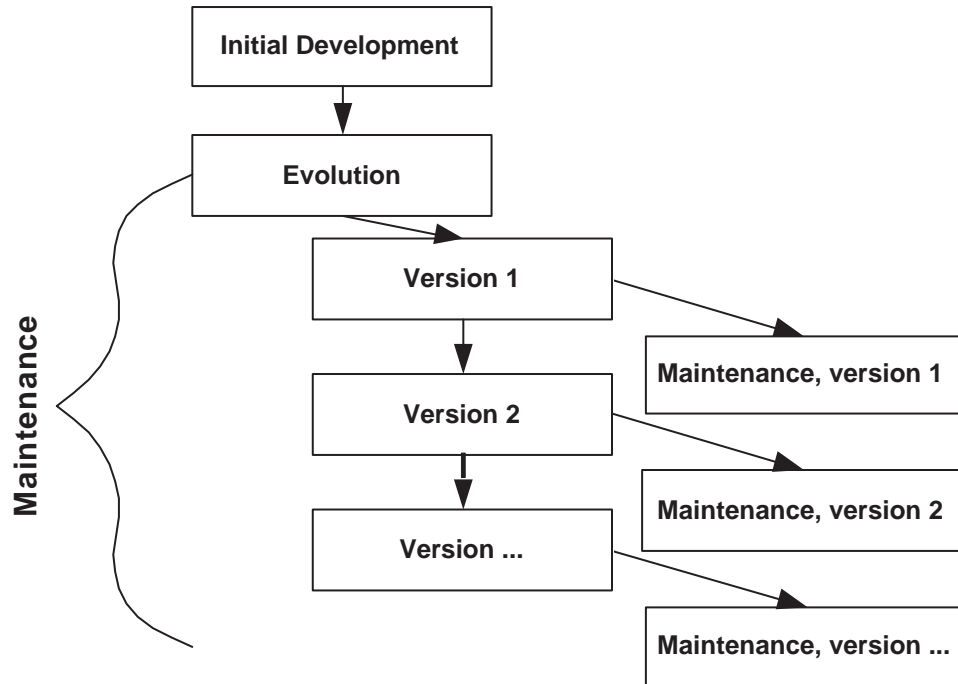


Figure 2.3: *Software Evolution in the Context of Software Maintenance*

2.6 Software Evolution Related Techniques

There is a number of techniques related to software evolution. All of them introduce means, which support the changes taking place in an evolution stage. Usually, those means aim at supporting the understanding and the reuse of assets from the previous development. Without covering them all, the next sections explore some of these techniques, which are important in the context of the thesis. These techniques are used and further developed in the presented work.

2.6.1 Program Transformation

According to [PTW]: *Program transformation is the act of changing one program into another. The term program transformation is also used for a formal description of an algorithm that implements program transformation. The languages in which the program being transformed and the resulting program are written are called the source and target languages, respectively.*

Software evolution uses program transformation. It is often the case, that a software system has to migrate from one development language to another in order to be able to fulfill the required evolutionary changes. Applying program transformation for evolutionary changes one makes reuse of the source code and adds to the system the features of the new development languages.

2.6.2 Reverse Engineering

According to the Reverse Engineering Taxonomy (IEEE Software, January 1990) reverse engineering is: *the process of analyzing a subject system with two goals in mind:*

- *Identify the system's components and their interrelationships;*
- *Create representations of the system in another form or at a higher level of abstraction.*

Reverse engineering is used when the knowledge and the documentation about a software system is lost. Reverse engineering aims at stating the base for application of other evolution-related techniques.

2.6.3 Refactoring

Martin Fowler defines in [Fowler 1999] refactoring as: *the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.*

Refactoring is a kind of reorganization, which is often applied when evolutionary changes are required in a software system. It is usually performed on small steps with small risk.

Refactoring is often applied to software architecture. The aim is to adapt the architecture to the latest understanding about the required separation of the problem domain concerns. Usually, an evolutionary change starts with architecture refactoring.

Refactoring makes reuse of both the domain knowledge integrated into a software system and of significant parts of source code.

2.6.4 Reengineering

Reengineering is defined in the Reverse Engineering Taxonomy as: *the examination of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.*

Reengineering and refactoring are very close in meaning. The difference is in the used methods of work. While refactoring is supposed to be done on small and more frequent steps, reengineering is to be done in one big step. Consequently, the reengineering risk is much higher. Reengineering is a sort of "revolution" for a software system. In that context reengineering is better applicable when talking about bringing a system back to an evolution stage, when evolving of the current system architecture is no longer possible.

Due to their close meanings, refactoring and reengineering are often mixed up in the practice. Nevertheless, the difference between them has to be clearly understood.

2.6.5 New Development (Replacement)

Another way to apply evolutionary changes to a software system is completely new development. The new development aims at establishing a completely new system, which only keeps the external interfaces and behavior of the old one.

New development is a technique applied only in case the required evolutionary changes can not be done through program transformation, refactoring or reengineering. This usually happens when system documentation and key experts are lost and the complexity of the system does not allow application of reverse engineering techniques.

While the previously mentioned techniques include reuse of the source code, new development includes reuse only of the domain knowledge.

Chapter 3

Methodology for Supporting Software Evolution

3.1 Motivation

Does software evolution need to be supported? Is the software development process not already far enough for such task? What is more to be done for the methodological support of software evolution?

The industrial background and the scientific research experience lead to the conclusion that the above stated questions still need some answers. Although it is certain that the contemporary software development requires software evolution, there is still a gap in its methodological supporting. The evolutionary changes, which add value to the software systems and keep them close to the business needs are always connected with a high risk and a lot of uncertainty. For many systems an evolution change is performed rather as "revolution", i.e. not as a planned constructive step. A lot of redundant work often accompanies the application of evolutionary changes. In many cases important design information is omitted during the changes, because of not methodological and unstructured work.

For all above listed reasons this thesis aims at establishing a methodology for supporting the evolution of software systems. The pre-stated goals are approached in two directions:

- Finding a method to support evolutionary changes in their most difficult part - changes of the system architecture;
- Finding a method for maintaining the system architecture and thus delaying its decay and postponing the need of revolutionary changes.

Denoted from the above statements, the work in this thesis is concentrated mainly on parts concerned with system architecture. The reason to do that is that architecture changes are the most important ones in evolution. The work is based on reuse of the knowledge, which is incorporated in the system in form of customer valuable and solution dependable features. These factors outline the topic of the thesis: It develops a "**Feature-Based Method for Supporting Architecture Refactoring and Maintenance of Long-Life Software Systems**".

Many of the ideas presented in this thesis are inspired from an industrial project, which was started in the year 2002. The project was about supporting a company in its activity to migrate its software product to a new system architecture. The cause for the decision to perform such a significant change was that the old system architecture was no longer capable to support new customer requirements and each change costed the company a lot of efforts and respectively money.

The project work revealed that the problems, which the studied system had were out of the scope of a normal software evolution. A small revolution within the software was required, otherwise the target system was endangered not to be able to take any evolutionary changes. The company has already performed such a small revolution two years before the project start. A lot investments were made during this phase, but the new system release was able to survive only for a short time before new significant changes to the system architecture were required. The company had invested a lot in evolving its product and was not able to profit from the investments for a long time. Meanwhile, the competition was able to improve its products, thus endangering the market position of the company.

The work on that project has proven the correctness of the presented research direction. It showed that the industrial software development requires a solution, which can help the software companies in:

1. Performing the software evolution in a way, which preserves or significantly extends the period between two revolutions. The companies are able to profit from their investments for a longer period of time.
2. The overall life-time of the software products is increased. The companies are able to keep their market positions for a longer period of time.

3.1.1 Preserving the Need of Software Revolutions or Increasing the Period Between Two of Them

The software evolution definition showed that the way to add value to the software products is by performing evolution changes. But there was an aspect of the definition, which required the changes to be done in a cost effective way, which means that it should be avoided that an evolution changes to a revolution. Revolution in a software product requires a lot of expensive investments. Reducing the need for revolutions in the life cycle of software systems can reduce significantly the price of the products. Performing evolution in a manner, which increases the time between two revolutions allows the companies to profit from their investments for a longer period of time.

One of the aims of the method developed in this thesis is to help increase the period of time between two large evolutionary changes in the software life cycle (Figure 3.1).

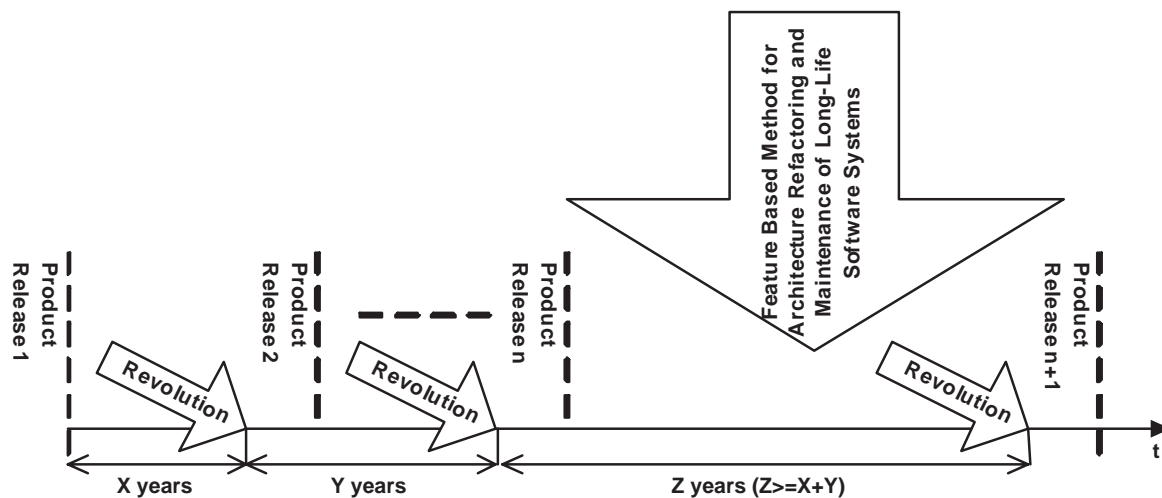


Figure 3.1: Increasing The Period Between Two Revolution Changes

3.1.2 Increasing The Overall Life-Time

The life time of software products depends on their ability to follow the changes of the problem domain. Products, which are easier to adapt to the changes serve customer needs better and respectively survive longer on the market.

Another aim of this thesis is to show how evolution can increase the overall life time of software systems. Products which evolve are much more competitive on the market.

3.1.3 Reducing The Maintenance Costs

Last but not least is the contribution to the solution of the classical problem of the high maintenance costs of software systems. This thesis proves that methodical support for software evolution results in higher quality of the evolutionary changes, which in return helps reduce the maintenance costs of the software systems.

3.2 Solution

The means, which are used for reaching the thesis aims are based on a single but very powerful asset, namely the knowledge incorporated within the software systems domain. This is a valuable asset, which has to be kept in an understandable and structured form and reused in every evolutionary change. This knowledge has been accumulated in the system over its whole life. This thesis shows how studying the domain knowledge helps making the right evolution decisions, especially the ones concerning architecture design. A presented set of techniques demonstrates how on the basis of the results of analyses of the domain knowledge, one can support and assure the consistency of the evolutionary changes, including system architecture consistency. The approach also shows how the reuse of the domain knowledge states the foundations of keeping the software architecture stable and up-to-date over a longer period of time.

3.2.1 Collecting, Structuring and Saving Domain Knowledge Embedded in Software Systems

Before a software system goes into a new evolution stage, the exact capabilities of the system must be known. All relevant knowledge about the system has to be collected, structured and saved in an appropriate model. This model has to give a comprehensive overview of the system features and the relations between them.

The above statement determines the first means, which is used for providing the searched solution. Feature models are introduced as a means to represent the domain knowledge embedded in a software system in a structured and comprehensive way. The thesis shows that feature models can be the necessary abstraction means, which fills the gap between system documentation and design and represents the system in an architecture eligible abstraction form. The presented work introduces a method, which contributes to reverse engineering and assists reverse engineers in their activities of identifying system components and their relationships.

3.2.2 Improving the Effectiveness of Changes

During the evolution stage, a software system is subject to many changes. It is important to perform these changes, but their effectiveness is also of great importance.

The domain knowledge collected in a feature model can be used in two ways for improvement of the effectiveness of changes during an evolution stage:

- Assisting system design in its architecture restructuring part;
- Assuring consistency of the system after architecture restructuring has taken place.

Assisting System Design

The most important activity when a software system switches to an evolution stage is the restructuring of its architecture. During architectural restructuring, there are two major types of modification: detection of architecture disproportions and their refactoring and detection of redundancies and their fusion. This thesis shows how the domain knowledge in form of feature models can be applied in these steps.

Two types of analyses are introduced in the thesis, namely statistical and information retrieval. Both utilize feature models and support architecture restructuring design decisions. The results of the statistical analyses result in a set of hints, which can help the establishment of a well-balanced architecture. The results of the information retrieval analyses can support the fusion of design redundancies and the identification of system components.

Assuring System Consistency After Changes

When something is changed, a common problem is to prove its compatibility with the old state. The same problem exists in architecture restructuring. After a new system architecture is established, it has to be verified that it is capable of satisfying all system requirements the old system did. In other words, one has to make sure the new system architecture did not "forget" some features along the way.

In the thesis it is shown how feature models can be a simple but useful mapping tool, which bridges between old and new architecture components. Feature models can support incremental design during architecture restructuring and can be used as a check point for the completeness of the restructured architecture.

3.2.3 Adapting the Software Engineering Process for Supporting Evolution

The evolution stage in the life cycle of software systems requires changes and correspondingly additional investments. In order to increase the benefits from the investments, the time when the systems evolve can be used as a good opportunity to improve the engineering process as well. The results of the evolution can state the base for the improvements.

This thesis shows and discusses a number of activities, which contribute to the software engineering process. The proposed steps utilize the assets of the collected domain knowledge and the obtained feature model.

Introducing Early and Low Risk Architecture Changes

Nowadays the evolution changes in a software system are not performed as fast as the problem domain requires it (Figure 3.2.). Usually evolution changes are postponed until the difference between the problem domain and the system architecture becomes so significant, that the domain requirements can not be further satisfied by the system architecture. This fact transforms the evolution changes into revolution ones, which are accomplished with much higher risk. The fact is mainly due to inconsistencies in the software engineering process. It is not capable to force the evolution changes as early as required from the problem domain.

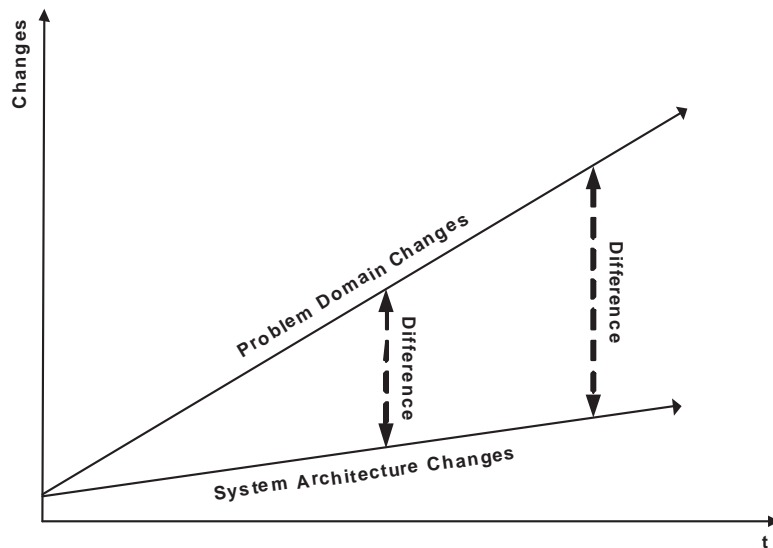


Figure 3.2: *Changes of Problem Domain in Comparison to Changes of The System Architecture*

The architecture decay can be delayed if architectural changes are made as early as possible. This thesis introduces a method, which operates in the requirements engineering phase of the software engineering process and forces early changes within the system architecture. In other words, the method assists maintaining system architecture up-to-date with the problem domain. The method operates on feature models and uses changes in them to force refactorings of the system architecture. Those refactorings help the system architecture to evolve together with the problem domain. As a result, a complete architecture restructuring is postponed.

Improve the Communication Between the Stakeholders

The good communication between the different stakeholders is one of the success factors for the software development. The industrial experience shows that features and feature models can be good means of

communications. The feature model obtained in an evolution stage can be introduced to all stakeholders and used further during the whole life cycle of a software system.

Introduce Common Domain Vocabulary and Means to Maintain it

There is one thing, which is crucial for the good stakeholders communication in the software engineering process, namely the common domain vocabulary. This thesis shows that a feature model actually contains such vocabulary. Updates to a feature model update the common vocabulary and vice versa.

Part II

State of The Art

Chapter 4

Software Life Cycle Models - Frameworks for Software Evolution

4.1 Motivation

Since the beginning of software industry, a number of models for the life cycle of the software products have been developed. The aim of these models is to provide a description ¹ of the events that occur between the birth and death of a software product.

Evolution ² is also one of the events, which occur in the software life cycle. Historically seen, a long time has passed before it has been recognized and explicitly described as such although evolution activities are present in any of the models.

In this chapter a short overview of some of the well known software life cycle models is presented. Attention is paid to the evolution aspects of these models. The presented list does not pretend to consider all available life cycle models; such a task is outside the scope of this thesis. The overview is performed considering the models, which present either important life-cycle modeling concepts or important evolution concepts.

4.2 Classic Waterfall Model

The first and the best-known approach to describe the life cycle of software systems is the *Waterfall Model* [Royce 1970]. The model was derived from other engineering processes and characterizes with its simplicity.

Waterfall model defines a number of sequential steps for development of software products, namely *analysis, design, implementation, testing and maintenance* (Figure 4.1). The software life cycle is split into three phases - *definition, development and maintenance*.

The waterfall model was accepted by management very well due to its simplicity and the well defined sequential activities; nevertheless, it failed to really match the needs of the software industry. The reason is simple. The waterfall model is actually no cycle model. It does not consider the fact that the software products have to evolve after the initial delivery, i.e. it does not describe software evolution as a separate phase. It can serve the development of a single software system well but it can hardly be applied to systems with long life time expectancy.

4.3 Incremental Model

In order to solve the problems concerning the need of the software systems to evolve, people have started to think about incremental models presenting the life cycle of software systems. In [Gilb 1988] an "Evolutionary delivery" model is described, which is also known as the *incremental model*. According to it, a software system has to be delivered in pieces, which have a priority, i.e. a software system is developed incrementally and the earlier increments determine the requirements for the later increments (Figure 4.2).

¹Some people prefer to talk about the life-cycle models not as a descriptive, but rather as a prescriptive.

²Evolution should be understood as it was defined earlier in section 2.4.

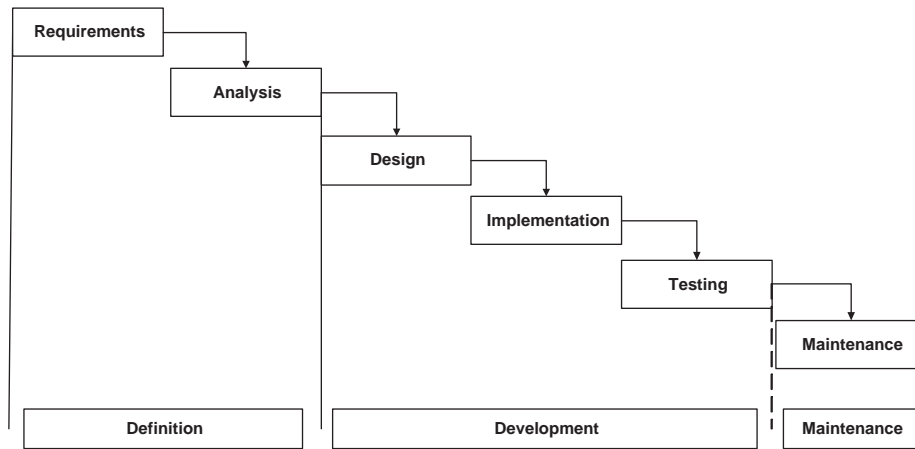


Figure 4.1: Waterfall Life Cycle Model

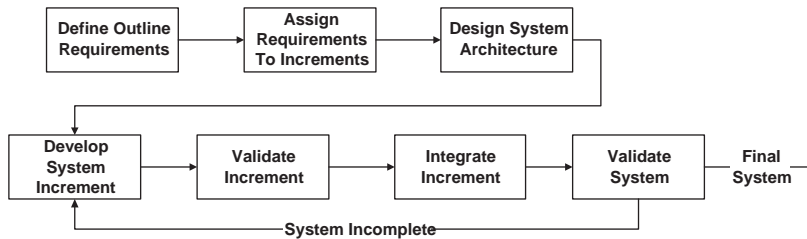


Figure 4.2: Incremental Life Cycle Model (adapted from [Sommerville 2001])

As can be seen from Figure 4.2, the incremental model splits the development phase of the waterfall model into an undefined number of increment producing phases. Each increment is the result of the evolution of the previous one.

The incremental model is a step ahead in comparison to the waterfall model. It considers the fact that the development of software systems is not a single but an evolutionary process. Still, the incremental model remains quite a theoretical approach when examining long-life systems. This is mainly due to the fact that the incremental model does not consider system architecture design as an incremental process. As it is shown in section 4.5, the main subject of evolutionary changes is actually the system architecture.

4.4 Spiral Model

In [Boehm 1988] the software process was presented as a spiral (Figure 4.3). According to the spiral model, the life cycle of software products requires a development process, which is not merely incremental. There is a new element, which precedes the sub-processes within the software life cycle spiral. This new element of the spiral model is the explicit risk analysis and assessment stage.

The spiral model views software evolution in a way similar to the incremental model. It brings the new element of risk assessment but, like in the case of the incremental model, it is difficult to apply the spiral life cycle to systems that are supposed to go through many changes during a long time after the delivery.

4.5 Staged Model

In [Rajlich et al. 2000] the authors K. Bennett and V. Rajlich present an empirical model for describing the life cycle of software systems. The major contribution of their work is that the well known software "maintenance is no longer described as a single uniform phase, but is comprised of several distinct stages, each with a different technical and business perspective". This new perspective is called *staged model*.

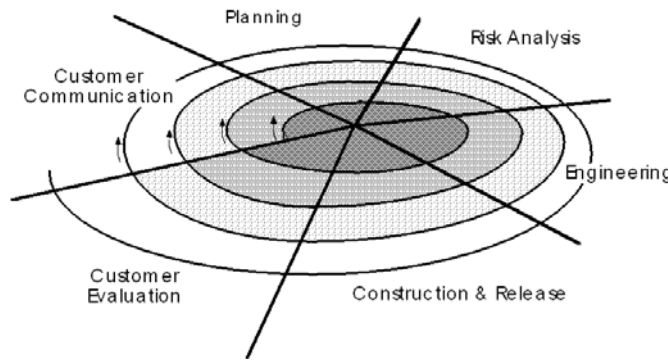


Figure 4.3: *Spiral Model [Pressman 1991]*

According to the staged model (Figure 4.4) the software life cycle consists of several distinctive stages:

- **Initial Development** - the first functional version of the system is developed.
- **Evolution** - the system is subject to changes, which extend the capabilities of the system in order to meet the user needs. The changes can be (and usually are) major ones.
- **Servicing** - the software is subject to minor changes, mainly repairs. No or small amount of new functionality is added.
- **Phase-out** - no more servicing is possible. The owners of the software system try to generate revenue as long as possible, but without any changes being applied to the system.
- **Close-down** - the software is withdrawn from the market and the users re-directed to a substitution system, if such one exists.

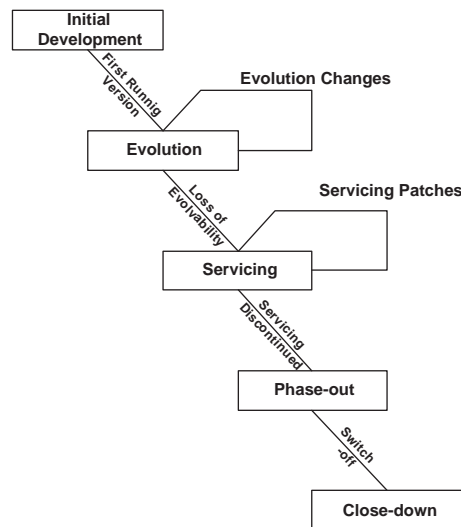


Figure 4.4: *The Staged Model for Software Life Cycle [Rajlich et al. 2000]*

The staged model is the first model for description of software life cycle paying special attention to the evolution of software products. According to it, evolution is a separate stage, which begins after the initial development has taken place. During the evolution stage "iterative addition, modification, or deletion of software functionality (program features) takes place", i.e. the software is subject to major changes. That formulation matches the reality of software products very well ³, especially for the ones with long life time expectancy. In the next chapters a number of technologies are discussed,

³This fact is actually not surprising, since the stage model is a result of an empirical study.

which have been developed to support the evolvability of software systems (e.g. domain engineering, software product lines, etc.). All these technologies propose software processes, which finally lead to a life cycle of the software products very similar to the staged model.

The authors of the staged model state that the main prerequisite for software evolution is the appropriateness of the architecture. When architectural integrity is missing, the software is no longer easily evolved, and only servicing is possible. Once a software system enters a servicing stage, it is very difficult (or even impossible) to go back to the evolution one [Bennett et al. 2000]. Very soon after the servicing stage, the software system enters the phase-out and finally the close-down stages. The staged model shows that if a company wants to protect its revenue, it should prevent the decay of the software architecture.

4.6 Agile Models for Software Life Cycle

During the last decade, a number of approaches have been developed, which present the development of software systems as an agile and mainly customer-oriented process. These are known as *agile methodologies*. These methods are based on a set of principles defined as the *Manifesto for Agile Software Development* [AM].

The agile methods determine a software products life cycle, which is quite different from the other known models. One can say that according to the agile approaches, a software system is permanently in an evolution phase until its close-down.

In this section the agile models are illustrated through a short overview of one of their main representatives, namely the *Extreme Programming* [XP].

Extreme programming (XP) was presented from Kent Beck in [Beck 1999] at 1999. Since then it is becoming more and more popular as a software development methodology because of its completeness and well performed realization of the agile manifesto principles.

XP defines a development process based on a set of simple rules and practices, whose interaction is expressed in a flowchart called *XP map* (Figure 4.5). The XP rules are divided into four groups, which represent the main phases of the development process. These are:

- planning,
- designing,
- coding,
- testing.

The planning phase is the one at which the user requirements are defined and a project schedule is established. The developing teams are also result of the work in this phase. After the planing is finished with the help of Class, Responsibilities, and Collaboration (CRC) cards the system design is made. The system design is continued in a coding phase, where the system is coded by units, which are consistently tested. The process continues with a testing phase, where an acceptance test is performed. The result of the testing phase is a release of the targeted system, which does not necessary cover all of the customer requirements. That is why the XP process does not end with the testing phase, but after the acceptance tests a new iteration of the XP development is performed. This makes the life cycle of the products developed with XP a continuous evolution, which ends only in case all of the customer requirements are satisfied or a system has to be closed-down.

Other representatives of the agile methodologies are Feature Driven Development (FDD) [FDD], Scrum [Martin et al. 2001], DSDM [DSDM], Adaptive Software Development [Highsmith et al. 2000] and Crystal [Highsmith 2002]. Although all these methodologies have shown a good potential for reaching faster evolving of software products and consequently quicker satisfaction of new customer requirement, they are hardly applicable to organizations where long-life and very complex systems are developed. This is mainly due to the fact, that the agile processes do not fit enough the structure of these organizations and have difficulties in providing the necessary communications within the organizations ⁴. Nevertheless, some of the principles stated by the agile models can be successfully applied for the evolution of long-life software systems as well. These are the principles for:

⁴This is actually a point where the agile methodologies should be explicitly strong. But this has turned out to be true in smaller organizations and development teams (10-20 persons).

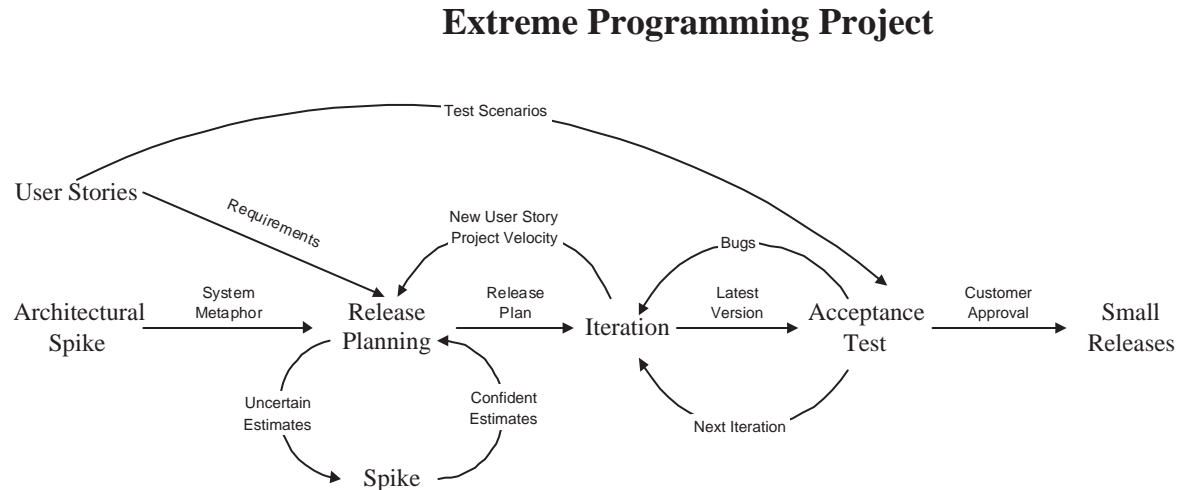


Figure 4.5: XP Map [XP]

- applying early changes to the developed products, even when these changes concern major system attributes, like the architecture;
- integration of business and development processes;
- direct communication between different stakeholders;
- "keep it simple" - the rule, which aims simplicity of the developed solutions and thus minimizing the development efforts and risk.

4.7 Conclusion

In this chapter a number of models have been considered, which describe the life cycle of software products. The presented models can be classified in two general groups, namely classical or conservative ones and modern agile methods. In sections 4.2, 4.3, 4.4 and 4.5 was shown that from the classical models, those which consider the software development process as an incremental one, are better in describing the software evolution, especially by systems with long-life time expectancy. The model best reflecting such systems is the staged model (section 4.5). According to it, software evolution determines a separate stage within the software life-cycle.

Section 4.6 made a short overview of the agile models and paid attention to the basic principles, which they present. Although the agile approaches are hardly applicable for evolution of long-life systems as a whole, some of their basic principles can be successfully applied in such systems. The principles of performing early changes, more direct communication between the stakeholders, simplicity of the developed solutions, integration of business and development processes break the clumsiness of the software development organizations, lower the complexity of the developed products and keep them closer to the application domain.

Chapter 5

Domain Engineering - Collecting and Implementing Knowledge for Reuse

5.1 Motivation

There are different ways to classify software systems. One of them is according to the business case they address, for example postal automation, point of sell services, airline reservations, etc. Another classification criterion is the implemented functionality, for example database systems, GUI libraries, workflow systems, real-time systems, etc. The areas referred around classes of systems or parts of systems are known as *domains*. Domains can be vertical or horizontal, depending on the classification criteria. Vertical domains concern the business case while the horizontal ones concern the type of implemented functionality (Figure 5.1).

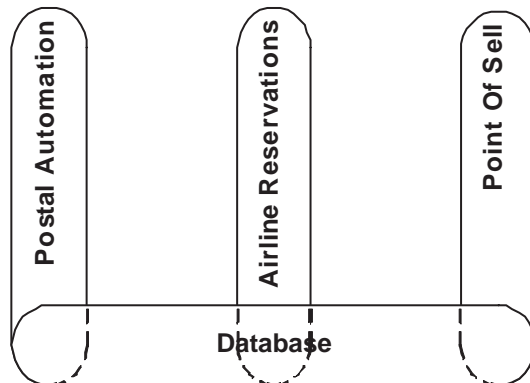


Figure 5.1: *Vertical and Horizontal Domains*

The software systems belonging to a common domain share many characteristics and requirements. These common traits present the domain knowledge. The reuse of that knowledge can obviously bring benefit to an organization able to manage it. Through capturing the acquired domain knowledge in the form of reusable assets and by reusing these assets in the development of new products, the organization will be able to deliver the new products earlier and at a lower cost.

Benefits from the use of domain knowledge can also be seen in the context of software maintenance and evolution. A well structured overview of the domain knowledge can contribute significantly when a software system is subject to reverse engineering. Usually, reverse engineering activities precede each evolution stage. It is almost impossible to accomplish them for large complex systems without deep knowledge of the systems themselves. The domain knowledge is the factor, which helps understanding and overcoming the complexity of the recovered information about the systems. Domain knowledge can support the design of a stable system architecture, which is a prerequisite of successful evolution.

At that point the knowledge about the commonalities and variabilities within a system and about the possible reusable assets can help the designers to establish the necessary abstractions. All these factors make the use of the domain knowledge obligatory in all maintenance activities.

5.2 In a Nutshell

The systematic approach for achieving the goals of using the domain knowledge is called *Domain Engineering*. According to Czarnecki et al.: "*Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable work-products), as well as providing an adequate means for reusing these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, etc.) when building new systems*". [Czarnecki et al. 2000]

This definition needs to be extended in order to match software evolution better. Domain engineering does not serve only the building of new systems; it also enables systematic use and reuse of the domain knowledge in order to support the establishment, maintenance and evolution of software systems.

Domain Engineering encompasses three main processes *Domain Analysis*, *Domain Design*, and *Domain Implementation*. The tasks of these processes are briefly described in the list below and discussed in detail in the next sections.

- Domain Analysis aims at identifying and defining a set of reusable assets for the systems in a domain.
- Domain Design aims at establishing a common architecture for the systems in a domain.
- Domain Implementation aims at implementing the reusable assets, e.g. reusable components, domain-specific languages, generators, and a reuse infrastructure.

The results of Domain Engineering are reused during *Application Engineering*, i.e. the process of building a particular end product in the domain (see Figure 5.2).

5.3 Domain Engineering and Software Evolution

If one considers the process of product development defined by the domain engineering approach (Figure 5.2) and the life cycle of the software products described by the staged model, (Figure 4.4) one can find a lot of similarities between them. The aim of the domain engineering process is to design and implement a base architecture for a number of applications, which is actually the initial development stage from the staged model. As the authors of the staged model state, it is rarely so that a system is delivered after the initial development [Rajlich et al. 2000]. The tasks of the initial development stage are to provide the base of evolving a software system. Later, during the evolution stage a number of deliveries can take place. The last statement forces an association with the application engineering process of the domain engineering approach. What the application engineering does is to use the results of the domain engineering in order to deliver an end software product and to force the evolution of the domain engineering results through introduction of new requirements.

Domain engineering has proven its capacity of bringing benefits by creating new software products. It is a process, which leads to a great level of software reuse and assures the robustness of the developed systems. The last statement is supported by a number of approaches, which are overviewed in section 5.7 and have been successfully applied in various domains. As it was previously shown, the domain engineering is also a process, which is very similar to the life cycle of long-life software systems, therefore one can conclude that software evolution can be supported by domain engineering methods. Domain engineering should be able to bring similar benefits to software evolution as it did for the development of new products.

The proof of the above statement is given later in this thesis. Many of the presented new ideas are based on use and reuse of the domain knowledge and the domain engineering methods. That is the reason why some more attention is paid to the domain engineering details. In the next sections, a short overview of some of the famous domain engineering approaches is performed. The concepts defined from those approaches and important for software evolution are outlined.

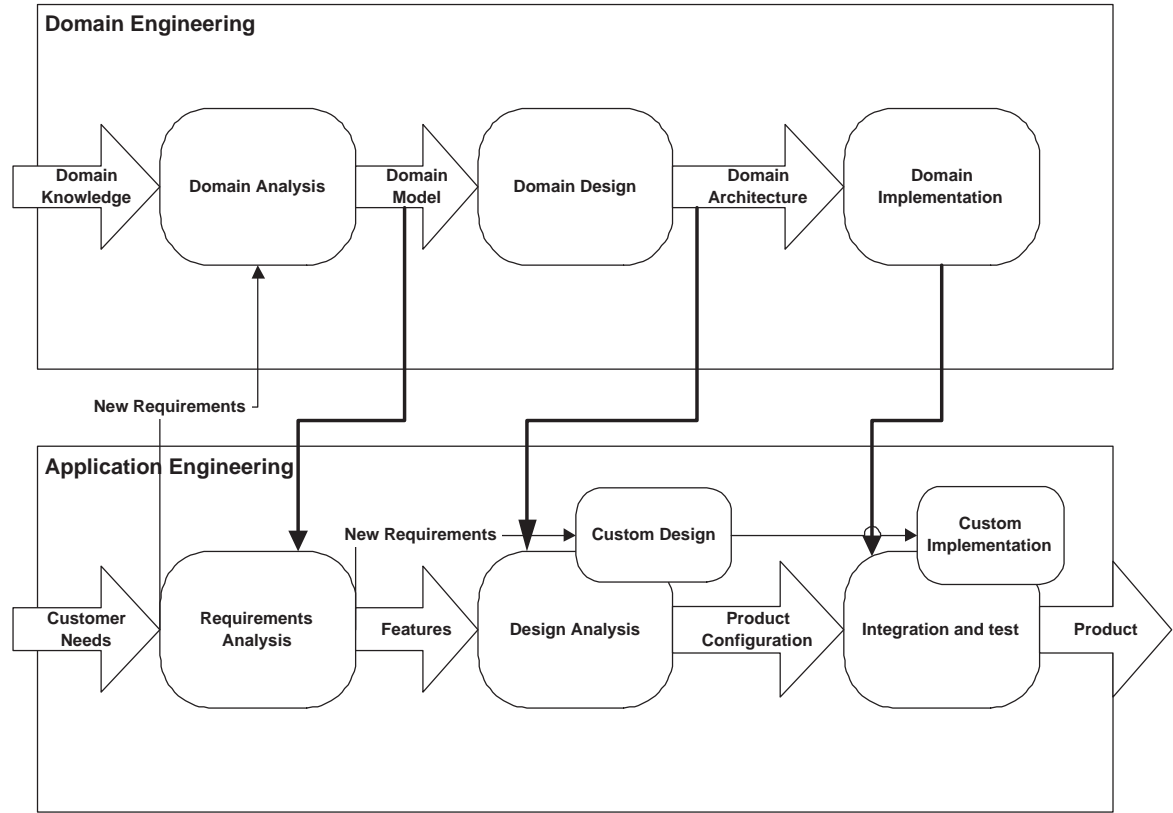


Figure 5.2: Software development based on Domain Engineering [SEIDE]

5.4 Domain Analysis

Domain analysis is the first process within domain engineering. Unfortunately, nowadays domain analysis lacks a common definition and understanding. According to [Kang et al. 1990], domain analysis is *"the process of identifying, collecting, organizing, and representing the relevant information in a domain, based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain"*. A more general definition, which is presented in [Fere et al. 1999] is: *Domain Analysis methods are used in formal reuse to identify, organize and model knowledge about the solution of a class of problems (domain) to support its reuse among all elements in the class.*

This thesis follows the second definition. It outlines well the tasks of the domain analysis that support the evolution of software systems. Briefly, the domain analysis should (Figure 5.3):

- Select, define and scope the domain focus;
- Collect relevant domain information (the domain knowledge) and integrate it into a coherent domain model.

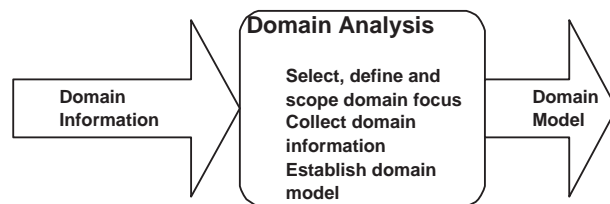


Figure 5.3: Definition of Domain Analysis Process [SEIDE]

The sources of domain information can include existing systems in the domain, domain experts, system handbooks, textbooks, prototyping, experiments, already known requirements on future systems,

etc.

The type of the examined objects, the analyses activities and the type of the resulting domain model depend on the reusable elements, which the domain analysis aims to identify. Some examples of different domain models are: domain feature model (FM), domain functional model (DFM), domain dynamic model (DDM), domain object model (DOM), domain information model (DIM), domain data dictionary (DDD), etc. A comprehensive overview of the different domain analysis methods is given in the studies of [Arrango 1994] and [Fere et al. 1999]. The authors have evaluated different domain analysis methods, focusing on the differences in the applied processes, the purposes and the results.

To some extent, all different domain models can support the software evolution. In general, as it is shown in the next section, the main task of these models is to provide a coherent input for software architecture design. But the use of the domain analysis results are not restricted to these cases. Later in part 3 is shown how some of these models, especially the feature model can be utilized for many tasks supporting evolution of software systems.

5.5 Domain Design

Domain design is the process, which uses the results of the domain analysis in order to develop a design model (Figure 5.4). According to [CARDS 1994]: "The key to design reuse is to develop and document an architecture that is generic enough for most systems in the domain and that supports the reuse of code components in the domain". The design model represents such generic architecture, which provides the basis for the development of the reusable components in domain implementation.

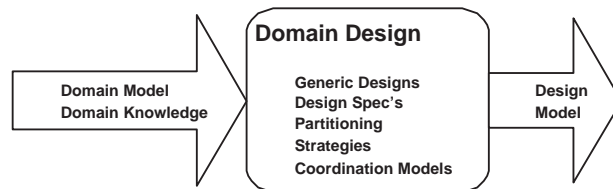


Figure 5.4: Definition of Domain Design Process [SEIDE]

As shown in Figure 5.4, there can be a number of activities, which take place during the domain design. These activities result in the following artifacts:

- The **generic design** aims at identifying an architectural style, which is most appropriate for the domain. This architecture style is also called **generic architecture**. The generic design also results in a partitioning strategy for allocating domain features to software elements and a coordination model describing how these elements are activated and share information.
- A **coordination model** defines the type of interactions allowed among software elements in an abstraction hierarchy and the mechanisms that support the interactions.
- A **partitioning strategy** defines the "legal" software elements (e.g. subsystems, objects, data types etc.) and how the domain features are allocated to them.
- A **design specification** describes the behavior of applications in a domain. This specification describes the internal operations identified in partitioning the domain information and specifies their associated constraints (e.g. definition of error conditions).

The results of the domain design are crucial for the success of domain engineering. The provided generic architecture is the basis of successful implementation of products within the domain. The ability of generic architecture to meet new requirements determines to a certain extent the ability of software systems to evolve.

In general, software architecture is an important software engineering paradigm and it is necessary to discuss it in more detail. This is done later in chapter 8.

5.6 Domain Implementation

Domain implementation is the process of identifying reusable components based on the domain model and generic architecture. Using the domain knowledge gathered during domain analysis and the generic architecture developed during the domain design, domain engineers acquire and, if necessary, create reusable assets, which are catalogued into a component library for use by application engineers (Figure 5.5). These reusable components, as well as application generators and domain languages, are the principal outputs of this phase of domain engineering.

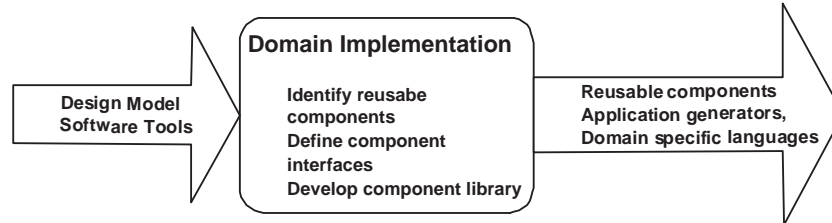


Figure 5.5: *Definition of Domain Implementation Process [SEIDE]*

The domain engineering process ends with domain implementation and, based on its results, a number of applications should be able to be established in an application engineering process. The practice shows that this number is not large. Of course, it depends on the types of the domains but they all evolve and the results of the domain engineering need to evolve as well. As figure 5.2 shows, the domain engineering activities need to be iterative. Usually, with a new iteration of the domain engineering the software systems go into an evolution stage.

5.7 Domain Engineering Concepts

A large number of domain engineering and analysis methods exist nowadays. There have also been published surveys of them [Wartik et al. 1992], [Arrango 1994] and [Fere et al. 1999]. Without attempting to go through all methods ¹ there is a need to pay some attention to certain concepts presented by some methods as they are used in this thesis.

5.7.1 Features and Feature Modeling in Feature Oriented Domain Analysis (FODA)

The Feature-Oriented Domain Analysis (FODA) is one of the most significant domain engineering approaches. It defines many important concepts, which are used in this thesis. FODA was developed at the Software Engineering Institute (SEI) as an approach, which defines a process for domain analysis and establishes specific products for later use. FODA comprises all previously described domain engineering processes. This section pays attention to the specific features, which FODA defines.

According to FODA the domain analysis process comprises three steps:

- Context analysis;
- Domain modeling;
- Architecture modeling.

Context Analysis

The FODA Context Analysis defines the scope of a domain that is likely to yield useful domain products. During the context analysis of a domain, the relationships between the "domain of interest" and the elements external to it are described and analyzed. The variability of those relations and the external conditions is evaluated. The kinds of variability to be accounted for are, for example, when applications in the domain have different data requirements and/or operating environments. The

¹It is definitely out of the scope of this thesis to make an overview of the domain engineering methods.

results of the context analysis, along with other factors such as availability of domain expertise, domain data, and project constraints, are used to *limit the scope of the domain*.

The resulting knowledge from the context analysis provides the domain analysis participants with a common understanding of:

- the scope of the domain,
- the relationship to other domains,
- the inputs/outputs to/from the domain and
- the stored data requirements (at a high level) for the domain.

Domain Modeling

For the domain that is scoped in the context analysis, the FODA Domain Modeling phase identifies and models the commonalities and differences that characterize the applications within the domain. It provides an understanding of the applications in the domain that is addressed by software. By systematically representing (or modeling) the functions, objects, data, and relationships of applications in the domain, domain modeling is used to define what the applications are, what the applications do, and how the applications work. The activities (or analyses) conducted during the domain modeling phase provide an understanding of the:

- features of the software in the domain;
- standard vocabulary of domain experts;
- documentation of the information (entities) embodied in the software;
- software requirements via control flow, data flow, and other specification techniques.

The FODA Domain Modeling phase produces a domain model, which consists of three components. Each component employs a separate analytical technique to model the interrelated components of the domain model. An information analysis produces an information model, a features analysis produces a feature model, and an operational analysis produces an operational model.

The information model represents what the applications are in terms of the entities, the feature model captures what the applications do, both in terms of operations and context, and the operational model relates the information model and feature model to the behavior and function of the applications.

The domain modeling process also produces an extensive domain dictionary of terms and/or abbreviations that are used for describing the features and entities in the domain model and a textual description of the features and entities themselves.

There is redundancy in the FODA domain models. For example, the information stored in the information model overlaps with the one provided from the hierarchy of the feature model. The terms contained in the domain dictionary are in many cases identical to the names of the features. Such redundancies cause maintenance problems and confusion by reusing the domain modeling results. In general, the most important model, in which the FODA domain modeling process results is the feature model. It concentrates the most important information gathered during the modeling activity. Later, chapter 10 pays special attention to this fact. Feature models are also used as the basic tool to support the new methods presented in this thesis.

Architecture Modeling

The architecture modeling phase of FODA results in an architectural model, from which detailed design and component construction can be done. The architectural model is a high-level application design. It focuses on identifying concurrent processes and application-oriented common modules and on allocating the features, functions, and data objects defined in the application models to the processes and modules. The architectural model defines the basic building blocks for an application. The architectural model defines the basic partitioning and interconnections necessary for constructing the application. This includes:

- model interface,

- model initialization,
- model execution,
- data base objects and
- the nature of the interconnections.

The architecture modeling phase of FODA reveals an important aspect of tracing features to architecture elements. Such traces are used later in part 3 and they are an important prerequisite for a number of techniques supporting software evolution.

The Concept of Features

The most important contribution of FODA, which is considered in this thesis is the presented concept of *features* and their modeling.

In FODA, features are the properties of a system directly affecting end-users: "Feature: A prominent and user-visible aspect, quality, or characteristic of a software system or systems." [Kang et al. 1990] For example, "when a person buys an automobile a decision must be made about which transmission feature (e.g. automatic or manual) the car will have." [Kang et al. 1990]

The features are modeled in a *feature model*, which consists of one or more *feature diagrams*. A feature diagram has the tree structure in which the root represents the concept being described and the remaining nodes denote features. An example feature diagram is shown on Figure 6.1. Within a feature model, the features can be marked as optional or mandatory. Constrains between features can be also established, for example inclusive or exclusive "or" grouping of features. Thus a feature model finally presents the common and the variable properties of a system affecting the end users.

Features and feature modeling are the basic instruments on which the work presented in this thesis is based. For this reason, they are discussed in more detail later in chapter 6.

5.7.2 Modeling the system dynamics in Reuse-Driven Software Engineering Business (RSEB)

The Reuse-Driven Software Engineering Business (RSEB) [Jacobson et al. 1997] is a model-driven approach, which aims at large-scale software reuse. The method has been developed on the basis of the experience of Hewlett-Packard Inc. (M. Griss) and Rational Software Corporation (I. Jacobson and P. Jonsson, formerly Objectory AB). RSEB utilizes the Jacobsons object oriented (OO) Software Engineering [Jacobson et al. 1992] and OO Business Engineering [Jacobson 1994]. Both technologies are applied in order to establish a set of reusable components, which will serve as basis for building a number of related applications.

The most important contribution of the RSEB approach is that it combines two significant analysis methods, namely the object oriented analysis (OOA) and the domain analysis. The RSEB can be considered an extension of the OOA with the domain engineering concepts and vice versa. This thesis considers the second point of view, namely: the important contribution of the RSEB of introducing the object-oriented concepts into domain engineering.

RSEB introduces use cases as a means to present domain knowledge. According to the approach, architecture and reusable subsystems are initially described by use cases and then transformed into object models that are traceable to these use cases. Use cases express the behavior of a software system in terms of functional requirements. Both people with and without technical background easily understand use cases. Unlike features, which make a rather static presentation of the capabilities of a software system, use cases contain the dynamics of system behavior. This is an important aspect, which contributes significantly to the consistency of the domain knowledge gathered during the domain analysis. It is shown later in section 10.4 that the aspects of domain knowledge presented by use cases are necessary for the successful evolution of software systems.

Another important contribution of the RSEB is the introduction of co-operative work between domain engineering and business engineering. As the authors of RSEB mention, software engineering needs to act together with business engineering in order to reach high levels of reuse. The same statement is true about the evolution of software systems. Later in chapter 13 is presented how the evolvability of software systems can be supported by changes to the business process. Some new ideas

are introduced, which concern requirements engineering process and related stakeholders. It is also shown how these new ideas can support software evolution.

Although RSEB itself is a great step ahead in the methodological development and evolution of software systems, its concept of domain analysis is incomplete. It lacks the benefits of the features. RSEB does not prescribe explicit models of the essential features, which characterize the different software versions and thus it is difficult to support sufficiently the evolution of software systems with the pure RSEB approach. The software evolution requirements can be satisfied in much greater extent through common use of the FODA and the RSEB approaches. The common use of features and use cases can lead to achieving significant improvement of software systems evolvability. This is proved by the FeatuRSEB approach, which is briefly examined in the next section.

5.7.3 The "+1" Model of Domain Engineering in the FeatuRSEB Approach

As it was mentioned in the previous section, the common use of use cases and features can bring a lot of benefits. Both present important aspects of the domains and one can say that they complement each other. The FeatuRSEB approach [Griss et al. 1998] proves this statement.

FeatuRSEB was developed at Intecs Sistemi in cooperation with Hewlett-Packard. It is the result of integrating the FODACOM approach [Vici et al. 1998], an object-oriented adaptation of FODA for the telecom domain, into RSEB. The FeatuRSEB approach introduces explicit domain engineering steps and explicit feature model to the RSEB in order to support component reuse. The targeted higher reuse rates also lead to higher evolvability of software systems.

The authors of FeatuRSEB put together feature and use case models and compare them according to the target audience they serve (Figure 5.6). As the authors state: "the use case model is user-oriented; the feature model is reuser oriented. The use case model provides the "what" of the domain: a complete description of what systems in the domain do. The feature model provides the "which" of the domain: which functionality can be selected when engineering new systems in the domain".

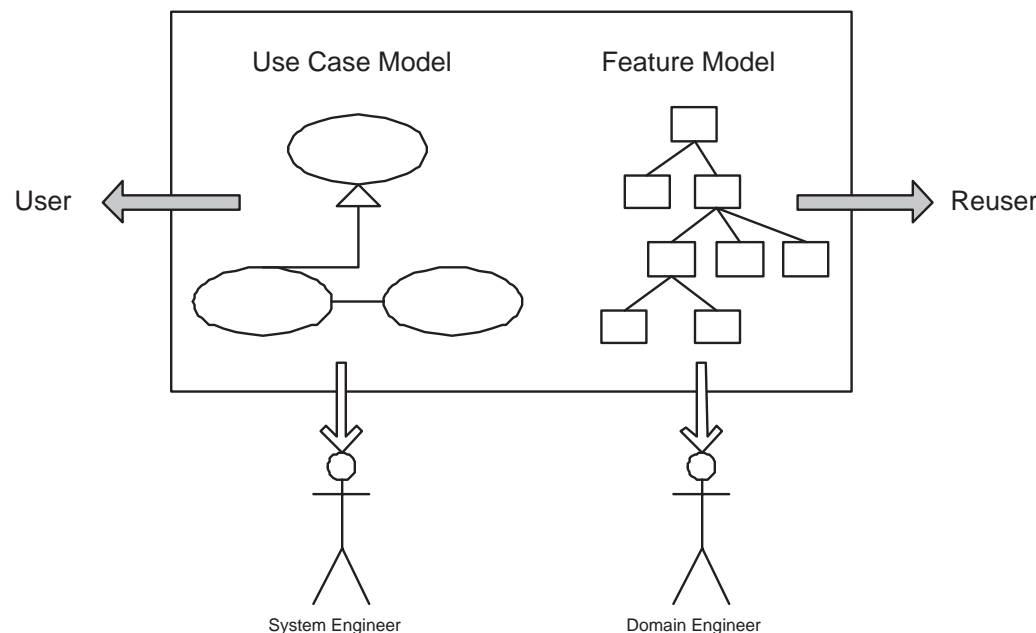


Figure 5.6: *Features vs. Use Cases* [Griss et al. 1998]

The result of the common use of feature and use case models in FeatuRSEB leads to the so called "+1" Model of Domain Engineering. The "+1" is the addition of a feature model to the original "4+1 View" approach to architecture (see section 8.3.1) used in RSEB. The feature model in FeatuRSEB replaces the use case model in the role of a unifier. Griss et al. note that the feature model serves "as a concise synthesis of the variability and commonality represented in the other RSEB models, especially the use case model." [Griss et al. 1998]

The FeatuRSEB approach is an important contribution in the field of domain engineering. The approach combines several modeling perspectives and dedicates them to the target audience whose

needs they serve best. Thus, all important aspects of domain engineering are revealed and modeled, which finally leads to improving the evolvability of software systems. FeaturSEB complements the successful architecture centric RSEB and adds to it the important feature model, which is the right means of presenting the commonalities and variabilities in a software system.

5.8 Conclusion

It was shown in this chapter that the application of domain engineering techniques helps reaching the goals of high evolvability and successful evolution of long-life software systems. As presented in section 5.3, the life cycle of such systems and the process described by domain engineering are very similar. In that sense, the concepts of the domain engineering technology can support software evolution in a natural way. This is the reason why, in this thesis the domain engineering methods are selected as a basis on which a new methodology for supporting software evolution are developed.

As shown in this chapter, domain engineering methods comprise three main processes, as each of them produces important results for software evolution:

- The knowledge collected during *domain analyses* is an important base on which software systems can be evolved. As described, domain engineering states two concepts for representing domain knowledge, namely use cases and features. Both concepts complement each other and their common use can increase software evolvability significantly. Later in part 3 of this thesis is presented a new methodology developed on the basis of these concepts. The new methodology comprises reuse of the domain information for reverse engineering tasks (chapter 11), analyzing the domain features in order to find and correct design problems (chapter 12) and optimizing the software engineering process by doing feature-based requirements engineering (chapter 13).
- In a *domain design* process a design model is developed, which is based on the models provided by the domain analysis. The design model contains the generic design and the strategies used further for the evolution of software systems. The design model is strictly linked to the results of the domain analysis, which means that traces between domain knowledge and design elements can be followed. This idea is later used in the newly developed methods. As described later, the traceability links between features and design elements allow analyses, showing design problems and hinting their solution.
- The domain implementation completes the foundations established by domain design and implements a reusable platform; this serves as a basis, on which a number of software systems can be successfully developed and further evolved. Nevertheless, this reusable platform should be kept up-to-date with domain changes. Unfortunately, the existent nowadays domain engineering approaches do not provide sufficient solution to this problem, which makes the development of a methodology for constantly updating the reusable domain platform a necessity. Such methodology is presented later in chapter 13.

Chapter 6

Feature Modeling

6.1 Motivation

Feature models were shortly discussed in the previous chapter. They were presented in section 5.7.1 as an important result of the domain analysis, which is later used in the domain design. Attention was paid to the fact that the feature models present the commonalities and the variabilities within a system domain. In section 5.7.3. was shown that feature models also present the reusable aspects of a domain, which is a missing aspect in other modeling techniques, e.g. use cases. Thus features and feature models seem to be in general good means for structuring and presenting the domain knowledge, especially by engineering for reuse and for evolution of the products over a long period of time.

Further more, chapter 10 shows how feature models are capable to present the system concerns in a structured way, which is very close to the actual architecture design. For this reasons feature models can play an important role in some evolution processes like reverse engineering and refactoring of software systems.

For all the reasons mentioned above, a full and comprehensive overview of the features and the feature modeling techniques, must be made. Such an overview is presented in the next sections.

6.2 Definition

Before one starts to study the feature modeling techniques, one needs to define what a feature and feature modeling is. For this purpose some of the definitions found in the literature are viewed:

- The feature oriented domain analysis method [Kang et al. 1990] defines feature as "an end-user-visible characteristic of a system". Examples of FODA features are features of a telephone switching system, transmission features of an automobile, etc.
- The organization domain modeling [Simos et al. 1996] methods presents features as "a distinguishable characteristic of a concept (e.g. system, component, etc.) that is relevant to some stakeholder of the concept".
- According to the rational unified process [RUP] features are "an externally observable service provided by the system which directly fulfills a stakeholder need. A property, like operation or attribute, which is encapsulated within a classifier, such as an interface, a class or a datatype".
- The feature driven development process [FDD] defines features as "granular functions expressed in client-valued terms using the naming template:

< action > < result > < object >

".

There also are another definitions of features (for example in [Griss et al. 1998]), but as it is visible from the above ones, there is unfortunately no unified understanding what the features are. The most comprehensive definition is the one provided from FODA, but for some reason it was not accepted directly from the other approaches. This is probably due to the fact that the domain modeling concepts,

which all these approaches use, have been developed in quite different ways. Despite that, the importance of feature and feature modeling requires a common understanding about the used terminology. A step ahead in this direction was the definition presented from Riebisch in [Riebisch 2003]:

"A feature represents an aspect valuable to the customer. It is represented by a single term. There are three categories of features:

- *Functional features express the behavior or the way users may interact with a product.*
- *Interface features express the product's conformance to a standard or a subsystem.*
- *Parameter features express enumerable, listable environmental or nonfunctional properties.*

"

The work in this thesis adapts the last definition of features. It is comprehensive enough and unifies in a certain degree the definition given by other approaches. This definition also serves good enough the aim of supporting software evolution.

6.3 Overview of Feature Models and Feature Modeling

Feature models are the means to structure the features and to express the relations between them. In general a feature model is a hierarchical representation of features. Usually the feature models are visualized in one or more feature diagrams. Feature modeling is the activity of creating the feature models.

Similar to the case about features, there have been presented many approaches for feature modeling and thus many corresponding definitions. A good overview of most of them was given by Robak in [Robak 2003]. This thesis does not make another overview of the modeling approaches, but there is a need for an introduction into the main concepts of the feature modeling.

For the presented overview of the feature modeling concepts the simple example shown on figure 6.1 is used. It is the classical "car example", which can be found in many literature sources. The structure of the example and the used relations will be explained in the next sections.

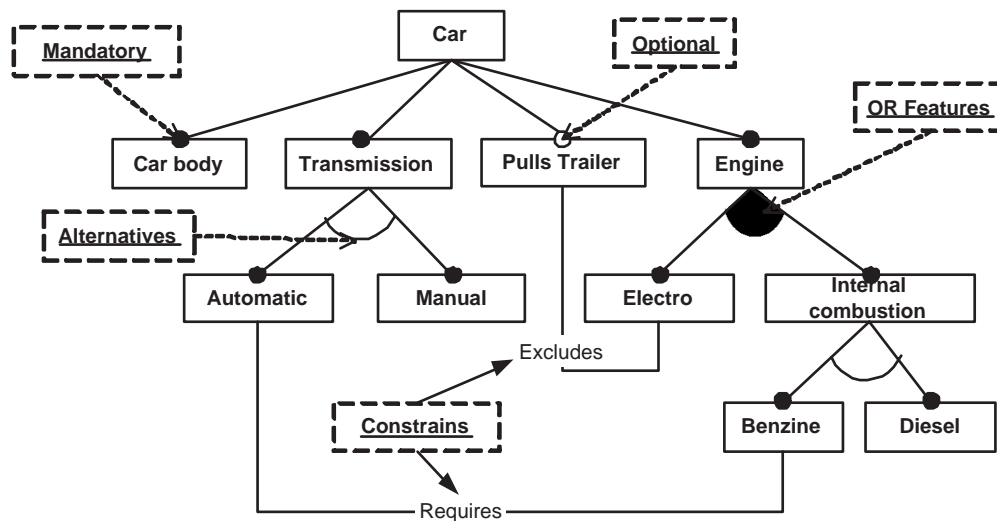


Figure 6.1: The Classical "Car Example" of a Feature Model

6.3.1 Feature Modeling Symbology

First little introduction of the used feature modeling symbology is required. In the literature there is a large number of feature modeling symbologies, similarly to the number of feature modeling approaches. For work in this thesis the symbology used from Czarnecki et al. in [Czarnecki et al. 2000] is utilized, which is based on the one presented from [Kang et al. 1990] and further in [Griss et al. 1998]. According to this symbology, there is a number of symbols, which can form a feature diagram and which are presented in Figure 6.2.








Nr.	Symbol	Meaning
1		Concept
2		Mandatory Feature
3		Optional Feature
4		Grouping of alternatives - XOR grouping
5		Grouping of options - OR grouping
6		Incompatible features
7		Required Features

Figure 6.2: Used Feature Modeling Symbology

6.3.2 Grouping of Features and Feature Hierarchy

As it is visible on figure 6.1 a feature model is a tree structure. The root of the tree is a special feature, which is called *concept* (Figure 6.2, symbol 1). Below the root, there can be an unlimited number of features, which are hierarchically structured. The hierarchy is not a random one, but as Riebisch defines: *"the position of a feature within the hierarchy shows its influence on the product line architecture"*. The hierarchical relations between features are also known as *feature - sub-feature - relations* [Riebisch 2003].

Within the feature hierarchy a feature can be marked as mandatory (Figure 6.2, symbol 2) or as optional (Figure 6.2, symbol 3). The mandatory features are the ones, which presence is obligatory in a software product. This means that no valid system can be configured without the mandatory features. The optional are not essential for the software products. Their existence is optional within a product configuration.

Features which have a common parent can be grouped by *multiplicity-grouping relations*. There are two types of such relations, namely grouping of alternatives or also known as XOR grouping (Figure 6.2, symbol 4) and grouping of option or also known as OR grouping (Figure 6.2, symbol 5).

The grouping of features was initially introduced from FODA as a way to express alternatives between two or more features neighboring in the feature hierarchy. FeatuRSEB has extended the approach by distinguishing between OR and XOR alternatives. Czarnecki et al. in Generative Programming [Czarnecki et al. 2000] have combined the OR, XOR and alternatives with designating the member features as mandatory or optional, i.e. he has introduced normalization of the feature groups. Common to all these approaches is that they can lead to ambiguities. This problem was solved in [Riebisch et al. 2002]. There the authors have introduced multiplicities in the feature grouping (Figure 6.3), which are similar to the multiplicities within an entity-relationship model. Additionally, the authors designate all features as optional in order to express the possibility of choice, but the choice possibilities are restricted from the defined multiplicities.

Although the approach to model feature groups with multiplicities avoids the ambiguities and seems to be a promising one, it lacks the corresponding tool support. That is the reason why this thesis keeps to the older symbology used in Generative Programming and FeatuRSEB. Later in chapter 14 a case study is described. During it a large number of feature diagrams have been created. This work would

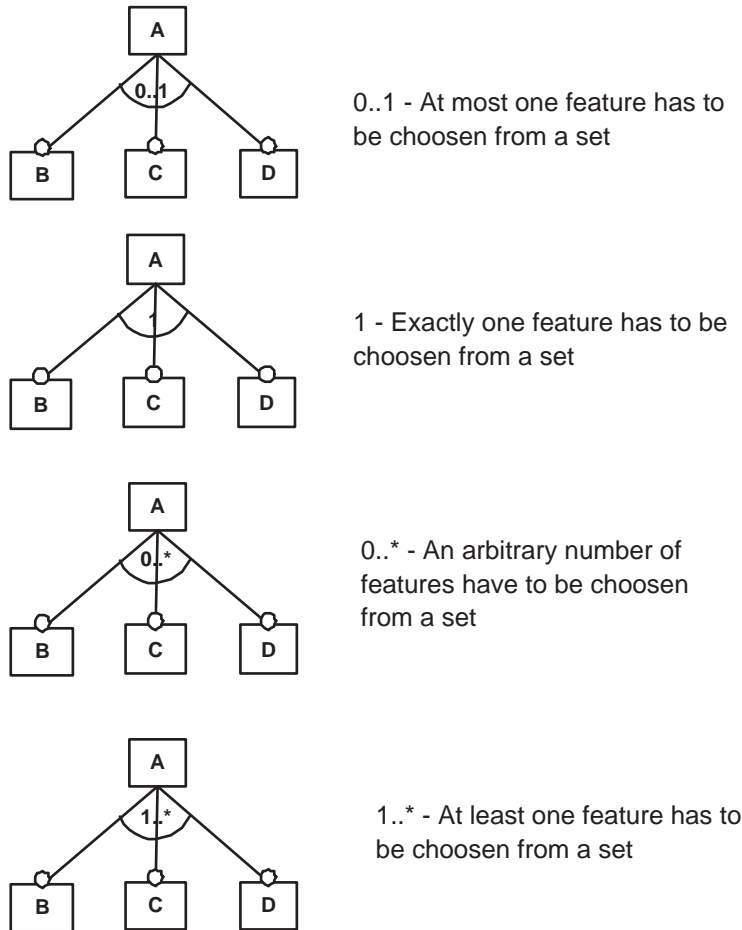


Figure 6.3: *Feature Model with Multiplicities*

not have been possible without a corresponding tool support.

6.3.3 Additional Relations Between Features

Additionally to the multiplicity-grouping relations there are other relations between features, which can exist in a feature model. Mainly, these relations express *constraints between non-neighboring features*¹. Such relations are:

- *Excludes relation* (Figure 6.2, symbol 6) expresses incompatibility between features.
- *Requires relation* (Figure 6.2, symbol 7) expresses "require" dependence between features.

The introduction of the additional relations within feature modeling is caused from the fact that a hierarchical tree structure is not always sufficient to present all possible relations between the features. There are cases of dependencies between non-neighboring features (features which are in different branches of the feature tree). For example on figure 6.1 the "Pulls Trailer" feature excludes the presence of the "Electro engine" feature. Both features are located in different branches and the normal neighboring grouping is not sufficient to express the relation between them.

There are some other additional relations, which are presented in the literature. For example Riebisch [Riebisch 2003] talks about a refinement and a hint relation. This thesis considers only the constraints expressing additional feature relations. The approach is to keep a feature model as simple as possible in order to be better understood from the stakeholders and thus to better support the evolution of software systems.

¹The use of the additional relations is not restricted only to the non-neighboring features, but is applied mainly to such features.

6.4 Conclusion

The expressive tree structure and various feature relations make a feature model a powerful means of modeling and presenting the characteristics of a software system. Furthermore, such a feature model is eligible to structure and expresses the domain knowledge in an understandable way. The simplicity and expressiveness of the model make it suitable for communication between stakeholders within the software engineering process. This is also stated in the software engineering institute. According to [MBSE97]: "The feature model is the chief means of communication between the customers and the developers of new applications. The features are meaningful to the end-users and can assist the requirements analysts in the derivation of a system specification that will provide the desired capabilities. The feature model provides them with a complete and consistent view of the domain."

Feature modeling is utilized in this thesis for supporting the evolution of software systems. The ability of this technology to concentrate important information about the system concerns and the abstractions which features provide, together with the clear graphical presentation provided by the tree structure of features, make feature modeling a powerful technology in the context of the software evolution. For the purposes of this thesis, the role of feature modeling in the software engineering process is further extended in chapter 10. In chapter 11 is shown how useful the feature hierarchy can be for the proper execution of some evolutionary tasks, like program comprehension, architecture recovery and architecture reconstruction. In chapter 13 is discussed, how feature modeling can contribute to the improvement of the software engineering process in order to prevent an architecture decay. And finally, feature models are recognized as the necessary communication means, which can be distributed through all stakeholders within software engineering.

Chapter 7

Re-Engineering - Analyzing, Understanding and Reworking Software Systems

7.1 Motivation

The main aim of the work presented in this thesis is to contribute to the evolution of software systems. Parts of this goal are approached by means of supporting software engineers in better understanding the code of the software systems, supporting the recovery of general information about the architecture of these systems and providing a number of hints, which can help restructuring of the architecture, in order to be able to satisfy the new requirements.

The syllable "re" has been frequently used in the above paragraph. It is present in the names of a set of methods, which are often used within the context of software evolution. These are the *re-engineering* methods.

Reengineering is defined in the reverse engineering taxonomy as "*the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form*" [RETAX].

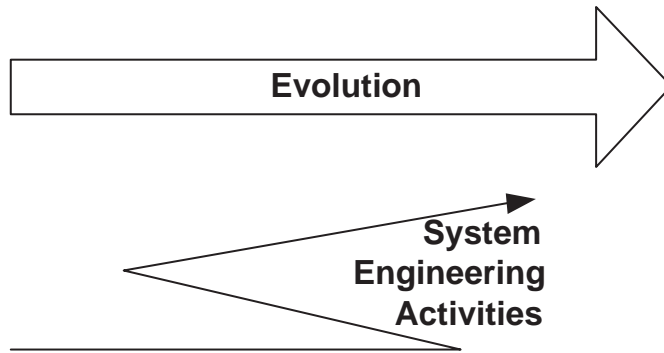
In other words, reengineering comprises a set of methodologies, which helps the understanding of a software system, the recovery of lost design information about it and finally restructuring and re-implementation of some of its parts. As it is shown on Figure 7.1 the final aim of all these activities is the evolution of the subject system. In general the "re" activities are probably the most significant part of the whole evolution process. For this reason it is important to get familiar with the basic "re"-engineering terminology and to provide an overview of some of the significant related approaches and to see their strengths and disadvantages.

7.2 Program Understanding

According to the UVic Reverse Engineering Tutorial [PTW] *program understanding*¹ is "the task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for software maintenance, software evolution, and reengineering purposes".

In a nutshell, program understanding is what is to be done, when a software system has to be understood. Usually one needs to build an own mental model of the system before he is able to understand it. Program understanding is the methodology, which helps the establishment of such mental model. As one can suppose such methodology is of a great use for maintenance and evolution of software systems. T. A. Corbi reports that up to 50% of the maintenance effort is spent on trying to understand code [Corbi et al. 1989].

¹A synonym of program understanding is program comprehension. The last one is also well known in the literature, e.g. [Program Comprehension. In the Encyclopedia of Computer Science and Technology, 35(20), Marcel Dekker, Inc:New York, 341-368,1995].



Reengineering supports evolution of software systems through performing backward understanding and forward restructuring and re-implementing activities.

Figure 7.1: *Reengineering* - adapted from [Nelson 1996]

The problem of program comprehension is quite an old one, it is supposed to be present since the dawn of the software industry. Many approaches have been published dealing with it. Most of them base themselves on analyzing the source code of the software systems. Although it is the most correct information source, it is often not sufficient due to its enormous complexity, especially in large systems. For this reason additional information sources are required. Such a source and probably the most important one, is domain knowledge. It has been proved that the success of the program understanding depends on the correct use of the domain knowledge [Rugaber et al. 2000]. In this thesis this idea is also supported and further developed by introducing a new way to present and to apply the domain knowledge (see chapter 11). But first, other program understanding approaches that make use of the domain knowledge are reviewed.

The authors in [Brooks et al. 1983] make for the first time the connection between program understanding and problem domain knowledge. They present a top down hypothesis establishment approach. According to it, when one understands a program, one has constructed a mental model of successive knowledge domains. Each of these domains consists of objects, properties, relations and operations. The succession of domains may include the problem domain, the domain of a mathematical model of the problem, the algorithm domain, the programming language domain, etc. One must also understand the relationships that exist between adjacent domains.

The ideas of the above described approach are elaborated in [Letovsky et al. 1986]. According to it, the task of understanding a program is one of uncovering the intention behind the code. Intentions are described as goals. Techniques for realizing goals in a particular implementation are called plans. Plans are similar to algorithms, but they may involve non-contiguous elements and may be combined in ways, which are not usually considered for algorithms. For example, two plans involving loops may be combined into a solution using a single loop implementing two distinct goals.

As mentioned, [Brooks et al. 1983] and [Letovsky et al. 1986] are top down approaches, i.e. their strategy is to understand the details described in a program source code based on a global domain knowledge. Such an approach is also the one presented in [Rajlich et al. 1994]. The authors extended the top-down program understanding techniques. The new element, which they present, is that according to them, a programmer creates a chain of hypotheses and subsidiary hypotheses, which are later verified in the code.

A common problem of the approaches described above, is the lack of a model, which presents the domain knowledge. This problem has been corrected in [Rugaber et al. 2000]. The authors show how a model of an application's domain is able to serve as a supplement to programming-language-based analysis methods and tools. A domain model contains knowledge of domain boundaries, terminology and possible architectures. This knowledge can help an analyst to set expectations for program content. Moreover, a domain model can provide information on how domain concepts are related.

All of the above mentioned approaches show the necessary relation between domain analysis and program understanding techniques. Still only one of these approaches gets to the conclusion that a

domain model should be used, namely Rugaber et al. 2000. Although the authors list a number of possible models, which can be utilized including an object oriented one, the experience gained in industrial projects requiring application of program understanding techniques shows that none of these models is simple and comprehensive enough to provide an easy to use design abstraction, which can be linked later with the source code of the studied systems. The work in this thesis solves this problems by introducing feature models as the models eligible for presenting the domain knowledge. Later in chapter 11 is shown how program understanding can be easily guided and supported by feature models.

7.3 Reverse Engineering

While program understanding is a technique, which primary helps the mental activities of getting into the source code of software systems, the *Reverse engineering* is a technique, which embraces activities for complete recovery of the system design and the system knowledge (Figure 7.2). Reverse engineering is defined as *"the process of analyzing a subject system with two goals in mind:*

1. *to identify the system's components and their interrelationships; and,*
2. *to create representations of the system in another form or at a higher level of abstraction*

" [RETAX].

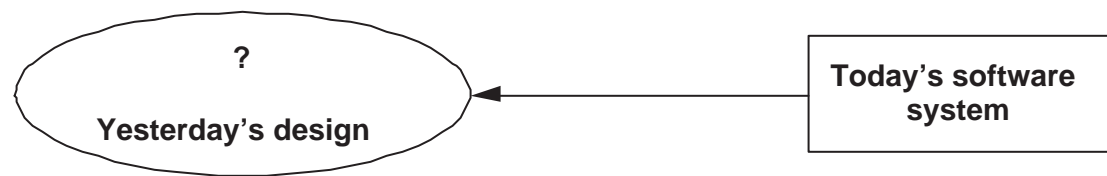


Figure 7.2: *Reverse engineering - adapted from [Nelson 1996]*

It is often the case that a lot of design information is lost after the initial development of a software system. Frequently, the documentation becomes quickly outdated and essential knowledge about the system is lost. This is especially true when persons leave the development teams. All these factors hinder the evolution of software systems and the support of reverse engineering methods is required.

The most important for the software evolution information, which has to be recovered with the help of reverse engineering techniques is the architecture design information. In the next section more attention is paid to *architecture and design recovery*. New ideas contributing to these techniques are presented later in chapter 12.

7.4 Architecture and Design Recovery

Design recovery is defined as *"a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system. The objective of design recovery is to identify meaningful higher-level abstractions beyond those obtained directly by examining the system itself"* [RETAX].

Design recovery can be separated in two main directions (Figure 7.3), namely *architecture recovery*² and *documentation generation*. The aim of the architecture recovery is to recover the software architecture, which represents the problem solution on a higher abstraction level than the source code (see chapter 8). Documentation generation is the act of deriving (on line) documentation from source code. The purpose is to help maintainers or developers to understand better the system they are working on.

In the context of software evolution architecture recovery is of great importance. The software architecture helps to reduce the complexity of the software systems and thus the complexity of the evolution tasks. A number of architecture recovery approaches are discussed in this section. Later chapter 11 presents a contribution to these approaches.

²Also known as architecture extraction

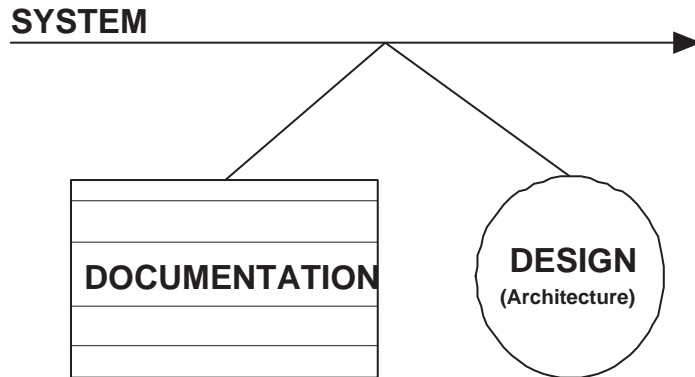


Figure 7.3: *Design Recovery - adapted from [Nelson 1996]*

The reverse engineering tool Rigi presented in [Storey et al. 1997] is used as a base for many architecture recovery approaches. Rigi is a graphical environment, which provides two contrasting approaches for presenting software structures in a graph editor. The first approach displays the structures through multiple, individual windows, while the second one, Simple Hierarchical Multi-Perspective (SHriMP) views, employs fish-eye views of nested graphs. Rigi is an open and extendable environment.

Based on Rigi is developed the "Bauhaus" one of the most significant architecture recovery approaches. "Bauhaus" is a project at the university of Stuttgart, Germany [BH03]. It aims at developing:

- means to describe the software architecture and
- methods and tools to analyze and recover the software architecture.

In more details, "Bauhaus" supports:

- derivation of different views on the architecture of legacy systems,
- identification of re-usable components, and
- estimation of change impact.

One of the important contributions of the "Bauhaus" project for architecture recovery is the introduction of methodology for tracing features into the source code of reverse engineered software systems [Eisenbarth et al. 2001]. This is a significant step ahead in applying the domain knowledge for architecture recovery, although the way in which features are used in "Bauhaus" environment is somewhat incomplete. The "Bauhaus" features are not structured and modeled, but considered as a linear list. This fact restricts the information provided from the features only to their functionality. Important aspects of the feature models, like the importance of the features for the system architecture, are omitted.

Another approach for architecture recovery, which is based on the Rigi environment is presented in [Riva et al. 2002]. The authors also apply the idea for tracing features into source code and thus discover architecture information. The new in the approach is that it pays attention to the dynamic behavior of the system, when a concrete feature is used. An extension of Rigi is presented, namely, support for Message Sequence Charts (MSC).

There are also other approaches, which have contributed more or less to solving the architecture recovery problem. As short overview to the ones presenting important ideas for the thesis is given in the next lines.

The workbench Dali [Kazman et al. 1997] helps reverse engineers in extracting, manipulating, and interpreting architectural information. Dali defines a framework for architecture recovery, a process and the needed tool support. "The architecture reconstruction process supported by Dali includes five phases:

- *View Extraction* - the purpose of the View Extraction phase is to extract information from various sources.

- *Database Construction* - the Database Construction phase involves converting the extracted information so that an SQL database can be created.
- *View Fusion* - the View Fusion phase combines various views of the information stored in the database.
- *Architecture Reconstruction* - in the Architecture Reconstruction phase, the main work on building abstractions and various representations of the data to generate an architectural representation takes place.
- *Architecture Analysis* - the Architecture Analysis phase involves analyzing the resultant architecture.

”[SEIDALI]

The idea of creating a database, where the information used for architecture recovery is stored and viewed is further developed in this thesis. The techniques presented in chapter 11 store the recovered information in a database of features, which is further used and analyzed for recovery of architecture information.

The Architecture Tradeoff Analysis Method (ATAM) [Kazman et al. 1998b] developed by the Software Engineering Institute (SEI) focuses into analyzing software systems with respect to specific quality attributes. ATAM also defines a set of steps and reusable architectural styles along with analytic models, that are used during architecture analysis. ”The output of an ATAM is an out-brief presentation and/or a written report that includes the major findings of the evaluation. These are typically:

- the architectural styles identified,
- a ”utility tree” - a hierarchic model of the driving architectural requirements,
- the set of scenarios generated and the subset that were mapped onto the architecture,
- a set of quality-attribute specific questions that were applied to the architecture and the responses to these questions,
- a set of identified risks,
- a set of identified non-risks.” [SEIATA]

The important contribution of ATAM in the context of this thesis, is that it identifies the need of discovering and modeling architecture driving requirements. These are usually the features and the non-functional requirements, which means that a feature model contains the architecture driving information.

The authors in [Finnigan et al. 1997] introduce the concepts of a reverse engineering environment called ”software bookshelf” as a means to capture, organize and manage information about a legacy software system. They distinguish three roles directly involved in the construction, population and use of such a bookshelf: the builder, the librarian and the patron. From these perspectives they describe requirements for the bookshelf as well as a generic architecture and a prototype implementation. The Finnigan et al. 1997 approach identifies and classifies the stakeholders, which can provide architecture recovery information. Interviewing these stakeholders can result in additional architecture information, which can be stored in a feature model and later reviewed from them. Such technique can assure an understandable and well structured presentation of the architecture information³, who’s correctness is additionally verified by system experts and thus assuring its correctness.

Bunch is a clustering tool, which was presented in [Mancoridis et al. 1998]. It creates a system decomposition automatically by treating clustering as an optimization problem. The authors present an automatic technique to decompose the structure of software systems into meaningful subsystems. The described automatic technique treats clustering as an optimization problem, that can be solved using hill-climbing and genetic algorithms, implemented in the Bunch tool. The idea of clustering the architecture information can be successfully developed with the help of feature models. In chapter 12 an approach is presented, which performs clustering analysis of the information stored in a feature model. Such analysis helps not only the extraction of additional architecture information, but also gives a base for future redesign activities by hinting the correct separation of system concerns revealed by the obtained feature clusters.

³Using the advantages of a well structured feature model.

7.5 Program Transformation

One of the common tasks for software evolution is to change one program into another. Since a software program is a structure with semantics, it allows its transformation into a new one. The semantics give the means to compare programs and to reason about the validity of transformations. For example software evolution often requires the application of new technologies, which are not available in the original program language. It means that the programs have to be transformed into another language in order to be evolved. The classical example of such a case is the programs written in the "C" language. "C" does not support the object oriented software paradigm. In order to use this paradigm a migration of "C" code into "C++" code is required.

The technique, which covers the above described transformation activities is called *program transformation*. The above presented example illustrates the most common application of the program transformation techniques, the language transformation. But program transformation is not restricted only to this area. One comes across it in many software engineering areas, for example: compiler construction, software visualization, documentation generation, etc. There are also many approaches for program transformation, which themselves have already become well known. A list of these approaches may include, but is not restricted to:

- Meta Programming [Cameron et al. 1994]
- Generative programming [Czarnecki et al. 2000]
- Program synthesis, refinement, calculation, etc.

There are two main scenarios for program transformation, which distinguish themselves by the types of the source and target languages. In the first scenario, the two languages are different and one can consider the program transformation as a translation approach. In the second scenario the source language is the same as the target language. In this case the program transformation can be considered as the act of rephrasing the source code. Both main scenarios can be refined into a number of sub-scenarios. Such refinement is done in the *transformation taxonomy* [PTW].

This thesis does not make an overview of the program transformation taxonomy, which comprises many concepts and it is disputable if all of them belong to it. Such task is out of the thesis scope, which concerns studying and developing technologies for supporting software evolution. It also is out of the scope of this thesis to discuss all program transformation concepts. Important for the thesis is the fact that program transformation is inevitably one of the activities for software evolution. It is expanded with other techniques (the so called "re" techniques⁴), which are more important in the current context and which are extended in chapters 11 and 12 in order to provide new methods, which support software evolution.

7.6 Redundancies and Clones Detection

Due to shortened development time, less mature development process and other reasons the software systems are not always build in the best way possible. In many cases the systems are released with a lot of redundant code and design redundancies. These problems usually are not severe for the current release, but become such when evolution of software systems is required. Most of the redundancies and the design problems are usually well known to the developers, but many of them are forgotten with the time. A special set of reverse engineering techniques has been developed for overcoming the problems of redundant development. These are the so called *clones detection* techniques.

The clone detection techniques have been precisely reviewed in [Bellon 2002]. The author has presented a comprehensive comparison of some of the most significant clone detection approaches. He evaluated them by considering the achieved values for the recall and precision metrics. The compared techniques are:

- Finding Duplication and Near-Duplication in Large Software Systems [Baker 2002]
- Clone Detection Using Abstract Syntax Trees [Baxter et al. 1998]
- A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code.[Kamya et al.]

⁴From re-verse engineering, re-engineering, re-factoring, re-construction, re-structure and others.

- Identifying Similar Code with Program Dependence Graphs [Krinke 2001]
- Advanced clone-analysis to support object-oriented system refactoring [Balazinska et al.]
- A Language Independent Approach for Detecting Duplicated Code [Ducasse et al. 1999]

Most of the techniques operate only on a source code level.

In this thesis some of the clone detection techniques are considered regarding their usefulness for supporting software evolution activities. Some of them are based on a full source code text view, while others focus on whole block sequences using metrics or pattern matching techniques. All approaches provide useful information about inherent clones.

In [Higo et al. 2002] the Gemini environment is described that can be used for analyzing the code clones and for modifying them e.g. for reducing the clone pairs and clone classes by using a so called Code Clone Shaper. In [Monden et al. 2001] the relation between software reliability and maintainability in connection with software code clones is shown. It is explained that modules having code clones can be more reliable on average than modules having non code clone. But modules having very large code clones are getting less reliable and are less maintainable. An approach for clone-analysis which focuses on the extraction of clone differences and their interpretation in terms of programming language entities and on the study of contextual dependencies is introduced in [Balazinska et al.]. This approach supports the computer-aided refactoring process and it is a good extension to our approach. Refactoring decisions can be done based on provided general information and special characteristics of selected clone clusters.

As it was mentioned, clones detection has an important place within the techniques used in software evolution. That is why there is a need to mention some of the most significant⁵ clones detection approaches. It is a common problem between all these techniques, that they work on the lowest design level - the source code. For this reason they tend to omit the design redundancies, which are caused by higher level design problems, for example architecture redundancies. Later in chapter 12 is presented an approach, which supplements the clone detection techniques and which deals exactly with the high level design redundancies.

7.7 Refactoring

An important part of the "re"-techniques used for evolution of software systems is the *refactoring*. Refactoring can be considered as the little brother of the reengineering. In general it is a technique having the same end effect, but using different strategy. Unlike reengineering, refactoring is performed often in a low risk steps. Refactoring has become very popular lately, especially with the introduction of the agile methodologies for software development (see section 4.6).

Martin Fowler [REF], [Fowler 1999] defines refactoring as *"a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"*. He continues and defines the verb *to refactor* as *"to restructure software by applying a series of refactorings without changing its observable behavior"*.

Refactoring is usually applied at the source code level, but in the context of large scale systems it can be an approach for overcoming problems within the system architecture. In this work refactoring is considered more as an architecture refactoring, than a source code one.

Typically refactoring is associated with:

- many small changes applied repeatedly,
- a catalog of widely discussed changes,
- explicit unit tests applied before and after each minimal change and
- interactive, perhaps tool-supported, but not automated process.

The approach presented in this thesis for methodological support of the software evolution is closely related to the refactoring ideas. The approach aims at lowering the need of complete reengineering of a software system by introducing timely refactorings of its architecture. Refactorings are less risky than reengineering and therefore they are more preferable for evolving a software system.

⁵According to [Bellon 2002]

Means to support refactorings are also introduced in chapter 12. Nowadays there are some approaches, which have the same aim. For example in [Hautus 1999] a tool is introduced that assists refactoring of source code in order to achieve a proper package structure. Therefore, they use a metric for evaluating the quality of package structure. The metric is defined as the weight of all desirable dependencies in all packages divided by the tool weight of the dependencies in all packages and provides a way to quickly evaluate the internal quality of large software products based on their source code. Similar to the work in [Simon et al. 2001], metrics are used in the methods presented in this thesis to detect a need for refactoring a given software system. Statistical Techniques can also be used to provide empirical measurement on the practical use of refactoring.

7.8 Conclusion

In this chapter a set of techniques has been considered, which are widely applied for software evolution, the so called "re" techniques. The most comprehensive approach is reengineering. It embraces full reconstitution of a software system in a new form and the subsequent implementation. Thus, reengineering for itself is a high risky and comprehensive activity. A number of supporting techniques are within the scope of the reengineering. Program understanding and reverse engineering are used for information recovery and building mental and physical models of the software systems at various abstraction layers. Architecture and design recovery aim at extracting of lost design information in order to use it in the evolution process. Program transformation helps software engineers to translate the program source code into a new programming language, in order to apply new technologies, which the software evolution requires. Redundancies and clone analysis show design and implementation problems, which reduce the evolvability. And finally refactoring presents an alternative of the reengineering, which helps the software systems to evolve with lower risk in smaller steps.

Later in this work a number of techniques are introduced, which complement and support reengineering and the related techniques. In chapter 11 a method for feature-based architecture recovery is presented. The method develops the idea of using the domain knowledge in reengineering tasks. The approach aims at recovering architecture and initial design by utilizing the advantages of the structured information stored in a feature model. In chapter 12 an approach for assisting architecture design activities is presented. The approach develops the ideas of detecting redundancies and previous design errors. For this purpose a set of analyses are performed over a feature model. The approach also develops the ideas of clustering the architecture recovery information. Thus a number of hints and clues for architecture designers are produced. Finally in the context of reengineering, the work presented in this thesis points out the need of migrating from reengineering to a refactoring strategy by evolving software systems. In chapter 13 a method is presented, which forces early architecture refactorings and thus prevents an architecture decay and the need of high risk architecture reengineering.

Chapter 8

Software Architecture - The Design Base for Evolution of Complex Systems

8.1 Motivation

The size and the complexity of software systems has increased significantly since the beginning of the software industry. The original software design problems like algorithms and data structures of the computation are no-longer the most heavy ones. Software development nowadays requires the solving of new kinds of problems, which are related more to designing and specifying the overall system structure. This issue embraces a set of sub-topics as, gross organization and global control structure, communication protocols, synchronization, physical distribution, composition of design elements, scaling and performance, etc. In other words, in order to manage the complexity of the software systems, it has become of a great importance to design first their gross structure with its building blocks and their dynamics and relations.

The above described design problems are covered by the architecture level of design or by the **software architecture**. Due to its growing importance and number of covered issues, software architecture is being considered as a paradigm. It has proved its vital importance for software engineering. Garlan et al. describe a set of reasons, which prove that effective software engineering requires facility in architecture design [Garlan et al. 1993]:”

- First, it is important to be able to recognize common paradigms so that high-level relationships among systems can be understood and so that new systems can be built as variations on old systems.
- Second, getting the right architecture is often crucial to the success of a software system design; the wrong one can lead to disastrous results.
- Third, detailed understanding of software architectures allows the engineer to make principled choices among design alternatives.
- Fourth, an architectural system representation is often essential for the analysis and description of the high-level properties of a complex system.

”.

A well designed software architecture can assure good evolvability of the software systems. As it was stated in the staged model [Rajlich et al. 2000], an evolvable architecture is a requisite for keeping a software system in an evolution stage. As long as the architecture of a system is no longer evolvable a system can be only subject to servicing.

Due to all the above described reasons, there is a need to pay more attention to the basics of the software architecture paradigm. Since software architecture and evolution are very close related, these basics determine also many of the activities performed for software evolution. In the next sections a number of approaches for doing software architecture are considered and important evolution concepts,

which these approaches state are noticed. Later on, these concepts are used and integrated into the work presented in this thesis for supporting methodologically software evolution.

8.2 Software Architecture Basics

The IEEE Recommended Practice for Architectural Description of Software-Intensive Systems [IEEE Std 1471-2000] defines architecture as: "*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*".

The above definition outlines several important aspects of the software architecture. First, software architecture consists of components and as Bass et al. mention it also defines components [Bass et al. 1998]. The architecture components can be a process, a library, a database, etc. The unifying factor between the different components is that they have *well defined and clearly specified externally visible properties and interfaces*.

Another task of the architecture is to specify the relations between the defined components. According to [Bass et al. 1998] "The architecture embodies information about how the components interact with each other. This means that architecture specifically omits content information about components that does not pertain to their interaction".

The architecture components and their relations outline an important evolution task, namely components identification by architecture design, redesign and recovery. The work presented in this thesis deals with the problems, which accompany these tasks. As it was already mentioned in chapter 11 an approach for architecture recovery is presented. A number of techniques, which support architecture design and redesign are presented in chapter 12.

Third and also important in the context of software evolution architecture task is that it defines the principles, which guide the design and the evolution of a software system. As it was mentioned many times before, the proper software architecture assures the software systems evolvability. The principle and the strategies, which architecture defines determine in a great extend the ability for software to evolve.

The ability to design good architectures requires many skills from the software architects, including talent. Nevertheless, with time a number of best practices has been collected, which solve standard design problems. These practices are known as *design patterns or architecture patterns*. Design patterns have been presented for the first time in [Gamma et al. 1995] and since then become widely known and used, especially for designing new systems. Since design patterns are the resulting work of experienced architects, e.g. [Garlan 2000], [Buschman et al. 1996], [Schmidt 2000] and have proved their effectiveness, their proper use increases the software systems evolvability a lot. Examples of design patterns are: Model-View-Controller (MVC), Observer, Interceptor, Object Factory, Bridge, etc.

Alongside with the design patterns a set of *architecture styles* have been developed. An overview of the common architecture styles is presented in [Garlan et al. 1993]. Both terms, design patterns and architecture styles overlap each other. The slight difference is that design patterns present rather a technical solution, while architecture styles present principle and guidelines for architecture solutions. Similar to the use of design patterns, the proper use of architecture styles determines the successful evolution of software systems. Examples of architecture styles are: Pipes and Filters, Client Server, Layered Systems, Event Based, Repositories, etc.

The identification of the proper architecture style and the use of proper design patterns in a software architecture is an important evolution task. This information should be recovered if lost. Such an approach is presented for example in [Philippow et al. 2003]. The authors describe a method for discovering of design patterns in an existing system. This thesis also contributes to the identification and the use of the proper architecture styles and design patterns. The feature-based methods presented in chapters 11 and 12 help the identification and documentation in forms of feature structures of particular design objectives and their related decisions. In that sense a design objective can cause the use of a particular architecture style or design pattern, which will be revealed in a feature model.

Since the scope and complexity of software architectures have increased, it has been necessary to introduce *architecture views*. An architecture view is: "A representation of a whole system from the perspective of a related set of concerns." [IEEE Std 1471-2000] There have been presented a number of approaches considering different architecture views. Two of them have become very popular due to their ability to serve good the architecture design. They are considered in the next section.

8.3 Software Architecture Modeling

Architectural modeling represents the framework for constructing an application. An architectural model is the high-level design of the application. It defines the applications basic building blocks. It also defines the basic partitioning and interconnections necessary for constructing the application. As it was shown in chapter 5, architecture modeling can be considered as a part of domain engineering. But in general, regardless of the use or not of domain engineering methods, the development of evolvable software systems requires applying architecture models.

In this thesis two of the fundamental works on architecture modeling are considered, namely the "4+1 view" model proposed by P. Kruchten [Kruchten 1995] and the "4 views" architecture model proposed by Hofmeister, Nord & Soni [Hofmeister et al. 2000]. Both models are selected due to their ability to serve the software evolution. The advantages and the disadvantages of the models are discussed in the next sections. Later in chapter 11 these two models are used as a frame, which determines the needed architecture elements discovered in an architecture recovery process. The "4+1 view" model is recognized as eligible to serve the architecture recovery of systems with object oriented source code and the "4 views" model is recognized as the proper base for architecture recovery by systems with functional or procedural source code.

8.3.1 The 4+1 View Model

The "4+1 View" model suggests organizing the architecture descriptions in five different categories called views (Figure 8.1): *logical view*, *process view*, *physical view* and *development view*. The fifth view, namely the *users view*, contains scenarios and use cases and is used for validation and unification of the previous four. The "4+1 View" model separates the static and dynamic aspects of software architecture.

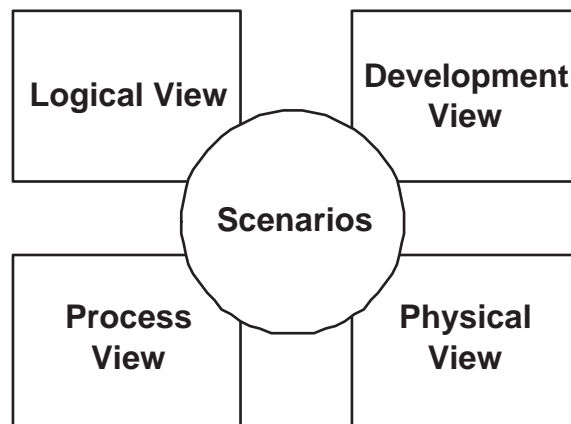


Figure 8.1: Architecture views in "4+1 View" model - adapted from [Kruchten 1995]

The logical view gathers the solutions of the functional requirements. It is tied closely to the application domain. In this view, the functionality of the system is mapped to architecture elements called conceptual components and the coordination and data exchange are handled by elements called connectors. For examples, the pipes and filters style may have filters as components and pipes as connectors. Since this view addresses the concerns of the end users, it usually uses domain terms and is independent from the software and hardware details.

The process view focuses on the dynamic aspects of the model and also describes runtime behavior. It also takes into account some non-functional requirements, such as performance, scalability, etc. A part of the process view is the execution one, which defines the runtime entities and their attributes. It also has components and connectors. The components in the execution view are tasks and the connectors are message, RPC, event broadcast and so on. The users of this view are mainly the system designers and integrators.

The physical view shows the solutions primarily to the non-functional requirements and also maps software to hardware. The work in this view is mapping various elements, e.g. networks, processes, tasks, onto the various nodes.

The development view focuses on the actual software module organization and on the software development environment. It also focuses on requirements related to the ease of development, software management, reuse or commonality, and to the constraints imposed by the toolset or the programming language. In this view, the components and connectors are mapped to subsystems or modules. It is focused on the actual software module organization on the software development environment. The software is packaged into subsystems and organized as layers.

The "+1" view collects scenarios, described in forms of use cases and sequence diagrams. The scenarios are in some sense the most important requirements. The use cases used in the scenarios capture the domain knowledge with its dynamics. The scenarios serve both as a driver to discover the architectural elements during the architecture design and as a validation and illustration role after design is complete. The scenarios should be specific. So a statement as "the system should be modifiable" is meaningless. Instead, one possible scenario about modifiability may be "it should be easy to add new features of the following type".

The "4+1 views" architecture model has become very popular during the last decade especially for new development. Compared with other methods, it is probably the most mature and widely used one. Each view has some specific notations or tools to support. An advantage of the model also is its full support by the UML [UML].

The "4+1 views" is a comprehensive approach, which also assures good basis for software evolution. It combines in a well defined manner the system design and the domain knowledge, which is an important prerequisite for systems evolvability. The approach also shows, that the domain knowledge, in this case presented in forms of use cases is the artifact, which unifies the different types of design information. Nevertheless, a significant disadvantage of the approach is that its models omit important information about common and variable system features, i.e. it misses a feature model. As it was shown in section 5.7.3 in discussing the FeatURSEB domain engineering approach, adding a feature view to the "4+1" views approach can improve its ability to support the establishment of evolvable software systems to a great extend.

8.3.2 The 4 View Model

The "4 views" architecture model (Figure (8.2)) is another well organized approach for modeling software architecture. It also proposes separate descriptions of the different architectural parts, i.e. several architecture views. These are: *conceptual view*, *module view*, *execution view* and *code view*.

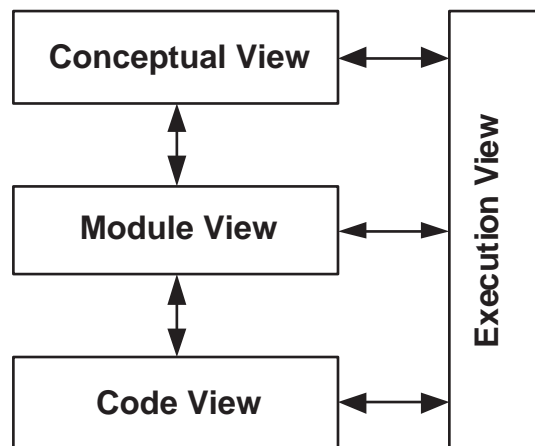


Figure 8.2: Architecture views in "4 View" model - adapted from [Hofmeister et al. 2000]

In brief the different views of the approach serve the purposes of:

- The conceptual view describes the system in terms of its major design elements and the relations between them.
- The module view presents the decomposition of the system and the partitioning of modules into layers.

- The code view is the organization of the source code into object code, libraries and binaries, then in turn into versions files and directories.
- The mapping from software to hardware and distribution of the software components is the task of the execution view.

The most important contribution of the "4 View" approach is that it defines the need of modeling the system concepts and the need of linking them later on with other more detailed design elements. The conceptual view is very close to the generic architecture, which is defined by domain engineering approaches (see section 5.7), which means that the "4 View" approach can be easily integrated with the domain engineering techniques. For example structures from a feature model can be easily linked with design concepts described in a conceptual view. This idea is further developed later in chapter 11, when a method for supporting of architecture reconstruction is discussed.

Unfortunately the authors of the approach do not go beyond of the software architecture problem and do not describe any possible cooperation of the "4 View" approach with other types of techniques, which makes it difficult to reveal the whole potential of the approach.

8.3.3 Other architecture modeling approaches

Additionally to the above described, view based architecture modeling approaches, there is a great number of others. This thesis does not perform a detailed overview of them, but there is a need to mention some of them in order to have a complete view over the important software architecture paradigm. A more comprehensive overview is presented at [Liao].

A full division in architecture modeling is determined by the *Architecture Description Languages* (ADL). The ADLs provide a formal description of software architectures. Typically the ADLs also provide tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions. Some of the significant representatives of ADLs are: Adage[DSSA-ADAGE:], Aesop, Rapide, SADL, UniCon, etc. These approaches are precisely discussed in [Garlan 2000] and [Clements 1996].

Due to their complexity the ADLs have not become popular. Nevertheless, if an architecture description with an ADL is present, it can ease evolution tasks such as architecture restructuring. The formal approach of ADLs allows the application of analyses, which can reveal architecture weaknesses and hint their correction.

The *Attribute-Based Architectural Style* (ABAS) approach explicitly associates a reasoning framework (whether qualitative or quantitative) with an architectural style [Klein et al.]. These reasoning frameworks are based on quality attribute-specific models, which exist in the various quality attribute communities. Each ABAS is associated with only one attribute reasoning framework. For those architectural styles that are interesting from different points of views, they may have various ABASs. For example, there may be distinct pipe-and-filter performance and pipe-and-filter reliability ABASs.

An ABAS is defined as a triple [Klein et al.]:

1. The topology of component types and a description of the pattern of data and control interaction among the components (as in the standard definitions).
2. A quality attribute-specific model that provides a method of reasoning about the behavior of component types that interact in the defined pattern.
3. The reasoning that results from applying the attribute-specific model to the interacting component types.

The idea of establishing a quality attributes model and relating it to a standard architecture solution is used in the approach for architecture recovery, which is presented in chapter 11. As it was earlier mentioned, in a feature model are stored both design objectives and decisions. There is a strong relation between the design objectives caused by non-functional requirements and the architecture quality attributes. And once again, the design decisions reveal the solutions, which are used for satisfying the objectives. For example a discovered design pattern is related to non-functional requirement like "portability" for example.

Architectural based *product lines* try to reuse the architecture design in a family of software systems and are proving to be a significant success for many organizations. When creating a product line, new challenges are encountered that do not occur in single product developments. First, in the product lines approach, one must consider requirements for the family of the systems and the relationships

between those requirements and the ones associated with each particular instance. In particular, the architecture for the product line should be easily instantiated or extended for new products. Second, the creation and management of a set of core assets is also challenging. This needs more work on the documentation and formalization.

Product lines are a natural approach for developing highly evolvable software systems. For this reason, they are discussed in more detail in chapter 9.

8.4 Conclusion

In this chapter the software architecture paradigm was discussed as the necessary design basis of complex software systems. It was also shown that architecture plays a great role in the context of software evolution. As it is stated in [Rajlich et al. 2000] the quality of the architecture determines in a great extent the ability of the software systems to evolve.

Software architecture modeling serves as a frame by performing different evolution tasks like discovering used architecture styles and design patterns and determining architecture components. The used architecture modeling method determines in a great extent the way of working by performing these tasks. Later, in chapters 11 a methodology for assisting architecture recovery is presented. This methodology requires an architecture modeling frame guiding the recovery process. For this purpose, the approaches "4+1 View" and "4 View" are considered.

The idea of the ALDs for performing formal description of software systems is used in this thesis. The work presented in chapter 11 is primarily based on the assumption that a feature model of a software system is very close to a formal description of its architecture.

The idea of establishing a quality attributes model of a software architecture and linking it to a standard architecture model has also found its application in this thesis. As described later in chapter 10, the feature models, which are the basis of all newly developed methods combine both design objective and design decisions. In some cases, the design objectives present necessary quality attributes and design decisions present their concrete solution, which in many cases is an architectural one.

Software architecture has been widely discussed over the last decade and as shown in this chapter a lot of work has been done on it. Nevertheless, the field for further development of the software architecture is still open. There are still plenty of open questions concerning mainly the criteria for finding proper architecture styles and architecture components. As a rule, a good architecture is the one making proper separation of the concerns in a software system. Unfortunately, nowadays architecture modeling and design count for this task mainly on the qualifications and the experience of the architecture designers. For this reason, later in chapter 12 is presented a method, which can hint the proper separation of system concerns and respectively assist the system architect's work. This method is a valuable add-on to the software architecture issue and as shown later its application can support well the solving of software evolution problems related to designing and redesigning software architecture.

Chapter 9

Software Product Lines - A Solution for Evolution of Long-Life Software Systems

9.1 Motivation

There is a very well known approach in manufacturing, which allows the companies to benefit from the common characteristics of their products. It is a common practice, a set of products that together address a particular market segment or fulfill a particular mission and share common properties, to be tied and their variability to be managed. Such an approach is known as *product line*.

Software product lines based on inter-product commonality are a relatively new concept that is rapidly emerging as a viable and important approach for software development. Product flexibility is the anthem of the software marketplace, and product lines fulfill the promise of tailor-made systems built specifically for the needs of particular customers or customer groups. A product line succeeds because the commonalities shared by the software products can be exploited to achieve economies of production. The products are built from common assets in a prescribed way.

Software industry can benefit from the product lines not only by marketing a set of related new products, but also through development of evolvable products. Similar to product lines, software systems, which are subject to evolution over a long period of time share many common assets among the different versions. In that sense, product lines are the natural way of supporting the software. They tie together and complement the assets revealed by domain engineering, software architecture and other software evolution related paradigms.

9.2 Software Product Lines Overview

In their work Kotler et al. describe product line as a "group of similar products" out of a specific problem domain [Kotler et al. 1999]. The approach is an extension of domain engineering with adding an accent to the establishment of a predefined architecture, which allows the derivation of a number of software systems. These architectures consist of common and variable parts. Variable parts can be changed or adapted to satisfy the special needs of an application. The means, which is usually used for managing the variability is feature modeling. Using generative techniques the different but similar products can be produced more or less automatically by product generators. The time for product delivery becomes comparatively very short. If there are new customer requirements that can not be fulfilled using the existing implemented features it is possible to extend the functionality of the product line. This leads to an evolutionary development of software product lines by extending the family requirements model, adapting the family reference architecture and implementing new components

The practice shows that the most usual way to establish a product line includes successful development of several similar applications [Koskimes et al. 1995] and reengineering of legacy software [Pree 1996]. These strategies are known as *evolutionary* ones. The evolutionary process starts from the conventional development process of one or more single application. Based on the experience from the previous development requirements for a system family are defined, introduced and later evolved

into a product line architecture. The development of the architecture is not only performed at conceptual level, but also at all other levels of the system development, such as requirements specification, design and source code. Evolutionary development takes place at each of these levels, providing two aspects: the content of the results on the one hand, and in its quality characteristics like maintainability, clearness and portability on the other hand. The process phases and activities are similar to those of software development in general, but they have to pay more attention to aspects like domain requirements, reuse and configuration ability (Figure 9.1).

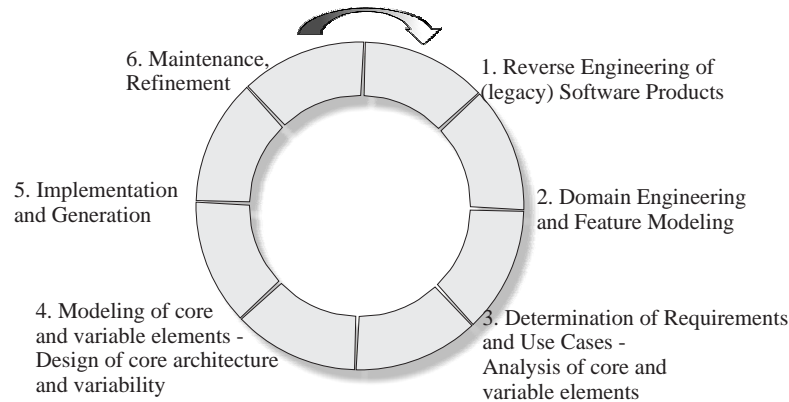


Figure 9.1: *Iterative Activities of the Development Process for Software Product Lines*

Another strategy for development of product lines is the *revolutionary* one [Boellert 2002]. In the revolutionary strategy a product line is introduced at once in a single step. The decision is usually made before the development of a number of planned systems.

In both cases, by the evolutionary and the revolutionary strategy, the management of domain information and family specific data is a key issue to the successful development of product lines. As described in [Ibanez et al. 1996] and in [Standish 1995] most of the current software development efforts fail because of a poor requirements engineering phase. For this reason, domain engineering methods and the product lines are very close related together. Boellert [Boellert 2002] has published a diagram showing the historical development of product lines methods, which illustrates well, the above statement (Figure 9.2).

As it is shown in Figure 9.2, additionally to the known domain engineering methods like FeatuRESB, FODA, etc., there is a number of approaches, which serve exactly the needs of product lines development. The most comprehensive ones are the developed at the Fraunhofer Institute for Experimental Software Engineering PuLSE [Boger et al. 1999] and Kobra [Atkinson et al. 2000] methodologies and the FAST methodology presented in [Weiss et al. 1999].

In general, product lines are an approach for software engineering, which embraces all important paradigms related to evolution of software systems. As it was shown, the product lines use domain engineering, develop evolvable architectures and perform reengineering of legacy systems. The cooperation of all these paradigms finally leads to an establishment of highly evolvable software systems.

9.3 Requirements Engineering for Software Product Lines - Driving the Software Evolution

One of the phases, which play an important role in the software development process and the correctness of its results determines in a great extend later the ability of software systems to evolve is the requirements engineering. Up to now requirements engineering was not considered. But in order to

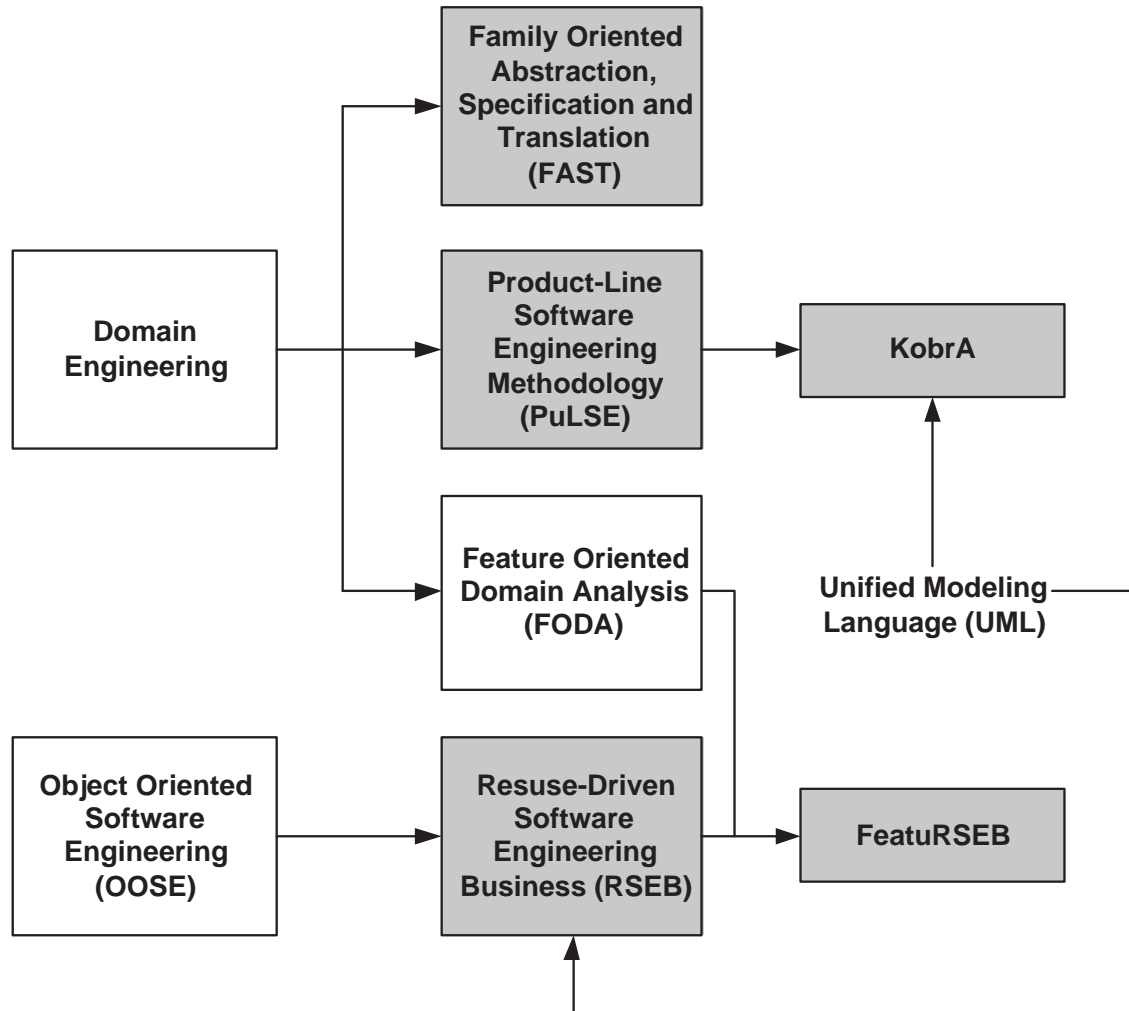


Figure 9.2: *Historical Development of Methods for Software Product Lines [Boellert 2002]*

reach the goals of establishing a methodology for supporting the software evolution, there is a need to make a short overview of it. Best is, to perform this in the context of software product lines, since the requirements engineering for product lines and evolvable systems are very similar. Later in chapter 13 new ideas are presented, which can help improving significantly the software evolvability by doing requirements engineering in a specific way.

Requirements engineering process is at the beginning of the software engineering chain. The correctness of its results determines in a great extend the success of the next steps. One can say that requirements engineering is a cog-wheel, which drives the next engineering steps. Unlike the case of developing a single software system, in the case of product lines or evolvable systems, the impact of requirements is not restricted to a single product but to a set of products or product releases. Thus the importance of a well-defined requirements engineering phase increases dramatically.

Traditionally, requirements engineering is divided into three main phases as described in [Hofmann 2000], [Sommerville et al. 1997] and [Loucopoulos 1995]. First, developers need to get familiar with the future product and its inherent problems. Within the elicitation phase these problems can be recognized by document analysis, interviews and prototypes. In addition, information on stakeholders, intentions, decisions or market surveys are part of the elicitation phase. All the information is evaluated to achieve the first milestone of software development, where a decision about continuing or discarding the project has to be made. Developers will link the pieces of information together to get a requirements model as the result of the modeling phase. The model contains different kinds of data. Simple textual notes, sketches, brief UML-models as an explanation of requirements, pictures and other data types which are useful for a good understanding of the ideas the future product is based on. The last phase of requirements engineering is the validation phase for comparing the product against the initial require-

ments. After all requirements are elicited, possible ways and procedures for their validation have to be added to the requirements model. Thus, a test strategy must be integrated from the beginning.

Requirements can be classified according to various criteria. A classification presented in [Kuusela et al. 2000] divides the requirements in two groups, namely *design decisions* and *design objectives*. All properties of the system related to the functional requirements are presented by the design objectives. The design decisions are a reflection of the solution domain to the requirements analysis, i.e. they capture the intention behind the designers decisions. This classification is considered later in the thesis, when the way in which the domain knowledge should be presented with a feature model is discussed. The idea of separating the design objectives and decisions is elaborated.

Activities for requirements engineering are embedded in the steps 1-3 of Figure 9.1. They have to be carried out in cooperation and connection with activities of the steps 4-6. To reach agreement about the future product elicitation, modeling and validation are processed with a number of iterations. The evolutionary development process of product lines is characterized by the following activities for reuse, refinement and improvement in addition to common software development [Philippow et al. 2001]:

- reverse-engineering and understanding former application architectures,
- comparing new requirements to the former ones,
- creating a new design, including both the new and the former requirements,
- redesigning the architecture and implementing new common and variable parts,
- documenting design decisions, intentions and the new architecture for future refinements.

For product line development the three phases of requirements engineering need to be adapted and extended to meet the special family needs.

- The concept of commonality and variability has to be laid down in the requirements engineering data model. New connections between the model elements and the family concept need to be established.
- Family development tools will have to support family specific views on the data model depending on the needs of the different stakeholders in the project.
- Analysis of the requirements model has to reveal family specific inconsistencies. For example, requirements with side effects to members of the family need to be rejected or costs estimations for the family itself and for each of the family member are needed.
- All the produced assets and data elements of the requirements engineering phase are interwoven and need to be related to the following development phases, which is addressed by the term traceability.

In chapter 13 an extension to the requirements engineering process is presented, which takes into account the specifics of the product line and the highly evolvable software systems. The contribution to the process is a methodology for initiating in the requirements engineering phase early changes to the system architecture, which adapt it to the changes of the problem domain. The presented methodology also satisfies the need of the product lines to reveal family specific inconsistencies and requirements with side effects to members of the family.

9.4 Conclusion

In this chapter the product lines approach for establishment of highly evolvable software systems was explored. It was shown how this approach unifies all important paradigms serving the software evolvability in a comprehensive way. It was shown how the application of product lines can support the evolution of long life systems, by assuring the correct use of all important assets in a family of related products or in a set of product releases. Nevertheless, the single application of product lines does not assure software evolvability. Reaching the software evolution goals requires further work. The development of long life and evolvable software systems needs constant support in many areas, including the correct comprehension of system requirements, the re-establishment of lost domain knowledge, designing proper architectures, which can be and are kept always up to date with the latest

changes in the system domain. The work in this thesis presents methods, which can assist these tasks. Although a product line is not required for application of these methods, its presence makes them much more effective. Due to the natural existence¹ of feature models in a product line the methods described in the thesis can be applied much faster and precisely, since all these methods utilize feature models.

Attention was also paid to the requirements engineering. As it was shown, it is a phase in the system development, which states the basis for establishment of system architecture. In the context of product lines, the requirements engineering results become even more important, since on this basis a family architecture is established. The results of the requirements engineering determine the success of the later developed architecture, which makes the process of great importance. Requirements engineering should be iteratively applied for all necessary changes in the systems family. It is the process, which can force the architecture adaptation to the latest domain changes and thus keeping the possibility of software systems to evolve. This thesis contributes to the requirements engineering. In chapter 13 a method is presented, which works in the requirements engineering phase and forces the necessary early architecture changes. Actually, an existence of a product line is not mandatory for the application of the method but makes it much more effective. This fact is also true for all other newly developed techniques, which are presented in this thesis. The application of all of them is much more effective in a product line environment, since most of the prerequisites for these techniques are existent in a product line.

¹In most of the cases

Part III

New approach

Chapter 10

Extending the Role of Feature Modeling in Software Engineering Process

10.1 Motivation

In the previous part of this thesis the feature modeling technology was presented. While discussing domain engineering and the software product lines approach, the potential of feature modeling for presenting reusable aspects of domain knowledge and modeling commonalities and variabilities within system domains was revealed. Although these tasks are of great use to software evolution, the application of feature modeling in that context should not be only restricted to them. Due to its capability of representing important domain information in a structured and synthesized way, feature modeling can be utilized in more aspects for software evolution. As an example, chapters 11 and 12 reveal how architecture recovery and assisted architecture design can be performed on the basis of feature models.

In this chapter, the basis of more intensive use of the feature modeling technology in solving software evolution problems is established. For this purpose, an overview of not satisfied evolution requirements is made. Later, it is shown how these requirements can be satisfied with the help of feature modeling. Thus, the technology is assigned an extended role corresponding to its potential.

10.2 Evolution Requirements for Using Domain Knowledge

A short overview of the evolution requirements for using domain knowledge is necessary in order to get acquainted with the aspects requiring greater methodological support.

As it was made clear in chapter 2, evolution is a continuous and repetitive process, which determines a separate stage in the life cycle of software systems. Later on - in chapter 5 - was shown that the correct application of domain knowledge is a prerequisite for a successful software evolution. So the first important evolution requirement is revealed: **a well structured and concise representation of domain knowledge is needed.**

In chapter 8, the software architecture paradigm was discussed and it was shown that it is the necessary important base of successful design and evolution of software systems. Later, in chapter 9 attention was paid once again to the importance of software architecture, but this time in the context of product lines. Alongside with the product lines, requirements engineering was also discussed as the process stating the necessary basis of developing software architectures. Unfortunately, although there is a strong relation between requirements engineering and architecture, there is also an abstraction gap between them. Practice shows that it is very hard to maintain correct traceability links between requirements and architecture. Sometimes requirements can be very abstract and architecture very detailed. And the opposite statement is also true. In many cases, it is difficult to trace an abstract architecture component to a concrete requirement. This is the reason why the existence of **a middle layer bridging the abstraction gap between requirements and architecture** becomes more important. The last sentence defines another evolution requirement.

In the next sections is shown how the two above requirements can be met with the help of feature modeling.

10.3 Structuring Domain Knowledge with Feature Models

How can feature modeling really support software evolution? In order to give an answer to this question one should consider the software developer's work. Features map really well to the way analysts or designers work. When analyzing requirements with the aim of constructing a system, one first looks for common properties. Thereafter, one tries to identify the variability of common properties. Having an overview of the common and variable properties, one might start to structure a targeting system, e.g. reusing already existing components that possess some of the properties, employing design patterns to develop new components, etc., making sure the resulting system has all the required properties.

These "intermediate properties" are obviously important, but they are seldom documented. They actually present knowledge of system domain and other related domains. Now, if the word "property" is substituted by "feature", a reason for introducing feature modeling in system design is revealed: *feature model represents design properties of resulting system and produces a structured documentation*. Furthermore, feature models can be used to capture certain design decisions. For example, a requirement states that an application should support changing the look-and-feel of the user interface, e.g. between Mac and Windows, without restarting. A design decision feature might be "provide an abstract GUI interface that supports varying the implementation at run-time." A developer might then search for a design pattern to come up with the "Bridge" pattern that will provide the required feature.

In general, one can say that a domain knowledge model is provided with a feature model, which, in addition to the commonalities and variabilities in a designed system, provides information about its **design objectives** and as it is shown later, information about its **design decision**. Thus, with constructing a feature model, the evolution requirement for well structured and concise presentation of domain knowledge is satisfied.

10.3.1 Retrieving Design Objectives by Analyzing Customer Needs

The term "design objectives" was already introduced in section 9.3. It was used to describe system requirements that define functionality. In the context of this thesis, it is a rather restrictive definition that needs to be extended. Obviously, the objectives in a software system are not merely its functional characteristics, but more. All customer needs, to which a target system has to provide a solution may be considered objectives. Some of these needs are often defined by non-functional requirements, for example "system performance". After analyzing customer needs, a set of system characteristics is established, defining a design frame for a target system, namely the design objectives.

How should design objectives be presented? It is important to have them structured and to be able to see the relations between them. Thus, performing correct separation of concerns will be possible, which finally means ability to define proper system components. Getting back to domain engineering methods, one can see that a result of domain analysis is a feature model. But analyzing the characteristics of a system domain and analyzing customer needs for a single system are two sides of the same coin. So a feature model should be the suitable means of presenting design objectives in a software system.

Figure 10.1 presents the feature model of an examined image processing system, which captures the design objectives of the system. As shown, each feature describes a characteristic of the system, which is related to customer needs. Next to the purely functional features like "Image binarization", "Image Rotation", etc., there are features like "performance" and "portability", which are added because of non-functional requirements.

10.3.2 Retrieving Design Decisions by Analyzing System Solutions

Together with the term "design objectives", the term "design decisions" was also mentioned in section 9.3. There it was used to describe system requirements, which capture intention behind designer's decisions. In this thesis, the word "requirements" is substituted in this definition with the word properties. Thus, in the context of this thesis design decisions are the system properties, which reflect the decisions taken to solve particular design objectives.

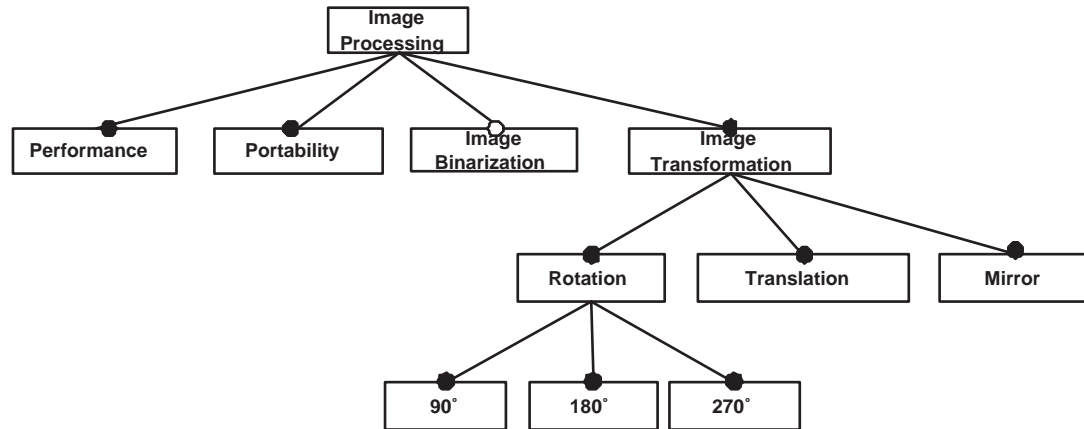


Figure 10.1: Design Objectives in a Studied Image Processing System

Design decisions are consequences of solving the design objectives and logically they belong together. In that sense modeling them together will constitute a comprehensive approach. In that sense, next to the design objectives, one can add design decisions in a feature model.

On figure 10.2 the sample design objectives presented in the previous section is extended with design decisions features. For example, performance-related design decisions are the "computing on demand" and the "multi-threaded computing" features.

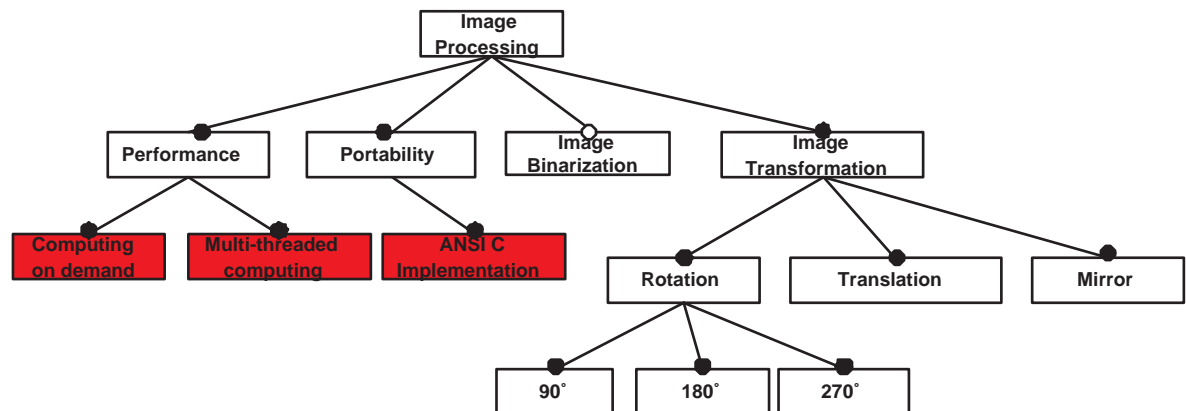


Figure 10.2: Design Decisions in a Studied Image Processing System

10.3.3 Making Overall System Presentation With Feature Models

By expressing both design objectives and design decisions in a feature model, a complete solution for making overall presentation of the important system characteristics can be established. Along with a feature model, a structure to the necessary domain knowledge is presented that later guides the design and the completion of software systems. Furthermore, expressing both design objectives and decisions with a feature model leads to the establishment of a basic unit, which constitutes the design foundations of systems. These foundations also present and document the minimum knowledge needed for future evolution.

If one examines the evolution of already developed software systems, one can see the need of structuring design objectives and decisions together in a feature model even better. Usually, such systems lack an overall presentation of their design history. It is often that such essential knowledge is completely lost due to various reasons. The recovery of this knowledge and its structuring is a prerequisite for further evolution of the systems. As one can see a feature model can be of great use in such cases. This problem is discussed in more details later in chapter 11.

10.4 Bridging the Gap Between Requirements and Architecture

Currently, the software process requires a direct relation between requirements and architecture (Figure 10.3). As already mentioned, this fact can cause an abstraction gap between them. In many cases, it can be difficult to trace directly requirements to architecture components and vice versa. As an example, four requirements concerning recognition tasks in a postal automation system can be taken, e.g. recognition of stamps, barcodes, addresses and logos. Upon analyzing the requirements, one can determine that all four tasks require processing of binary images, and that three of them require processing of grey-scale images. Thus, two mandatory features of the system have been identified: processing of binary and grey-scale images. Now, components processing both types of images can easily be designed and implemented.



Figure 10.3: *Current Relations Between Requirements and Architecture*

It was also shown in the previous sections that one might put both design objectives and decisions in a feature model. As it was defined, the objectives are revealed upon analyzing customer needs and the decisions upon analyzing system solutions. As a result, it is possible to unify elements related to requirements and to solution space in a single model.

For all the above reasons, in this thesis a suppression of the direct link between requirements and architecture and introduction of feature models as a medium is proposed (Figure 10.4). Feature models are recognized as a means eligible to satisfy the evolution requirement for a middle layer, which bridges the abstraction gap between requirements and architecture.

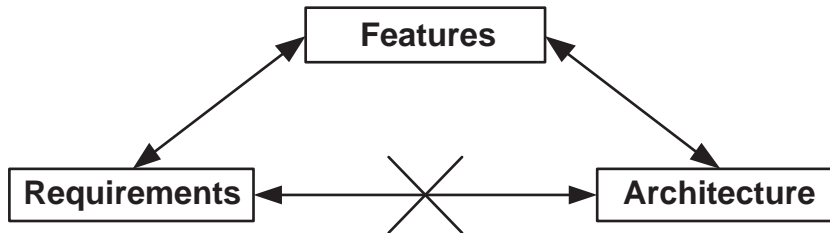


Figure 10.4: *Breaking the Direct Relation Between Requirements and Architecture*

After the introduction of features in the software engineering process, the following relations can be defined:

- Features - requirements relation.
- Features - architecture components relation.

There is no one-to-one connection between requirements and features, or between features and architecture components (Figure 10.5). Each requirement calls for one or more features, which are potentially referenced by other requirements as well. Likewise, one or more components might be responsible for implementing a certain feature. Note that features may spawn new, derivative requirements for the components, and a decision has to be made during requirement engineering on the component level as to how these should be documented.

Nevertheless, for the sake of simplicity of the approach presented in this thesis, the multiplicities in the above listed relations are restricted (Figure 10.6). The features-requirements relation is restricted to one-to-n relation. That means, *a feature groups a set of requirements*. Similarly, the features-components relation is restricted. In the context of this thesis, it is n-to-one. That means *a component implements one or more features, but the possibility of implementing a feature in more than one component is excluded*.

Although the above restrictions reduce the capacity of modeling, they have two benefits, which are considered important, first: they **increase the simplicity of the approach** and, second, they

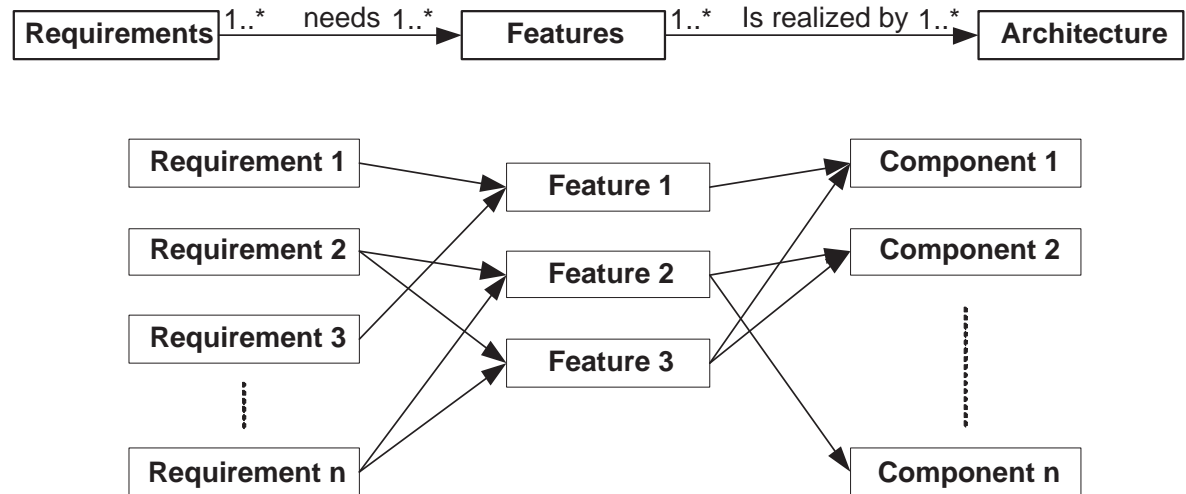


Figure 10.5: Common Relations Between Requirements, Features and Architecture

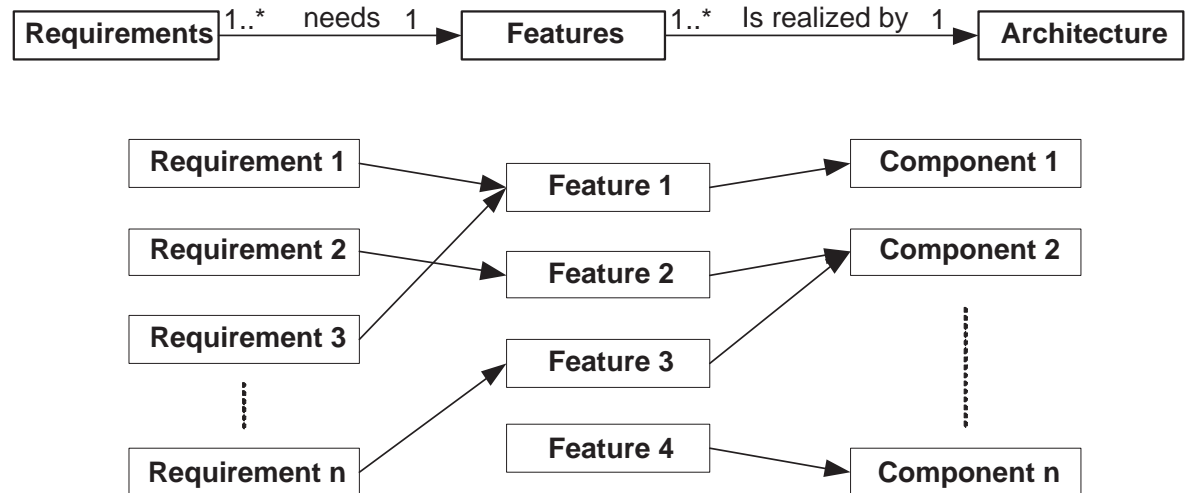


Figure 10.6: Restricted Relations Between Requirements, Features and Architecture

introduce harder discipline in the software process. The last point is particularly important in order to reach more evolvable software systems.

10.5 Combining System Dynamics and Feature Models

Earlier, in section 5.7, the domain modeling concepts were presented that are important for software evolution. Use cases were also discussed and the significant role that they play, especially by common use with feature modeling, was mentioned. It was shown how feature models and use cases complement each other to provide a comprehensive overview of system characteristics.

This thesis also considers use cases as participants in the software evolution process. Use cases are viewed as units that complement a feature model with means of expressing the dynamics of software systems. As it was shown in sections 5.7.2 and 5.7.3, the use cases themselves are not sufficient for building a model supporting software evolution but their common use with feature modeling can be a comprehensive approach. This is the reason why in this thesis some design objectives presented in a feature model are related to use cases. Figure 10.7 presents what these relations look like in the previously mentioned studied image processing system. As an example, part of the feature model is used, which describes recognition tasks, called "reading".

As shown in figure 10.7, features needing more detailed description of their functionality are related

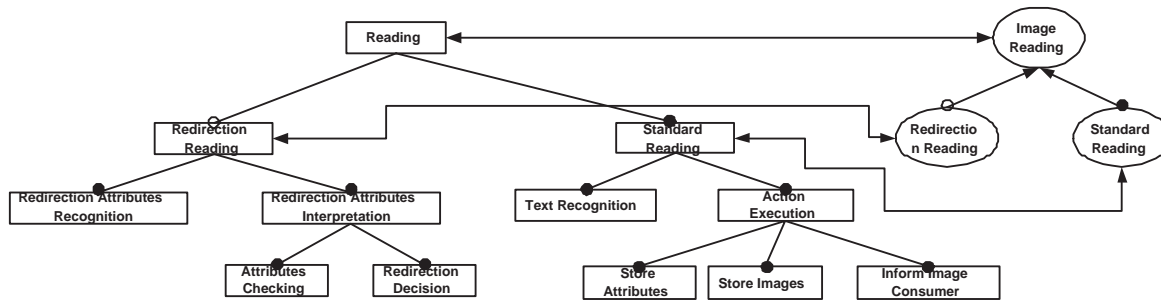


Figure 10.7: Relating Features and Use Cases

to use cases describing this functionality. Usually, such features are located at the top of the features hierarchy. This is due to the fact that in the common case, when a feature model of an existing system is constructed and the features from the model have to be related to components from the system architecture, the hierarchical structure of the feature model corresponds in a great extent to the hierarchical structure of the system components. Thus, the features belonging to a higher hierarchical level normally comprise more functional characteristics than the ones located on a lower one.

10.6 Overall Structure of an Evolution Friendly Feature Model

By describing the role that feature models can and should play in software evolution it was also shown that the models are supposed to have a certain structure assisting this role. Features describing design objectives are supposed to be on a higher hierarchical level than features related to design decisions. On the other hand, features that need to be described in more detail contained in use cases are also supposed to be located on a higher level than features, which are meaningful for themselves. Thus, a certain structure of feature models can be reached and defined, which is to be used to support software evolution activities.

Two main hierarchical levels of features in a feature model can be defined, namely:

- **Level 1 - features related to use cases and/or to non-functional requirements;**
- **Level 2 - features providing the details of use cases and/or non-functional requirements.**

According to the design objectives/decisions classification, one can expect to find on the first level of the hierarchical structure of feature models predominantly design objectives. The second level is to a greater extent dedicated to the design decisions, although the presence of design objectives features should not be excluded. Figure 10.8 illustrates an example of the above-described structure. Again, the example involves extraction from the feature model of the studied image processing system.

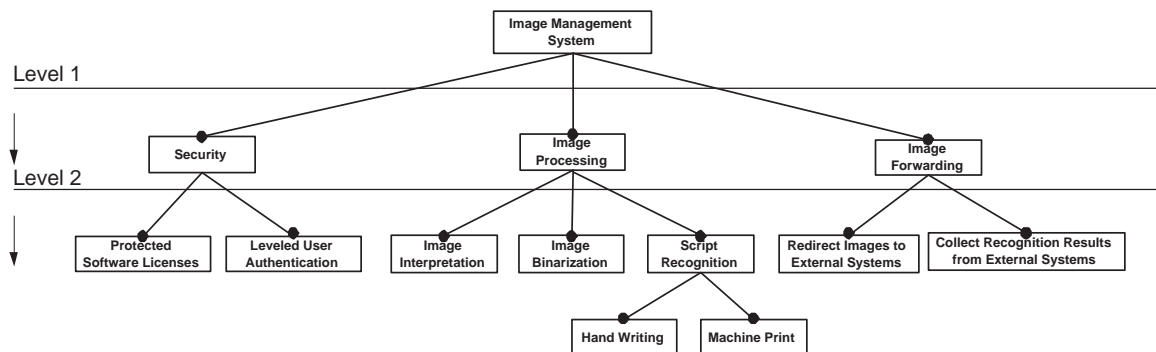


Figure 10.8: Example of a Feature Model with Layered Structure

The example on Figure 10.8 is structured as follows: The top node of the feature model presents a concept feature, as it is supposed to be by the definition of feature modeling. Below the concept feature,

there are features that present two main use cases of the studied system, namely: "Image Processing" and "Image Forwarding". Those two features also present the main objectives of the studied system. On the same level, a feature is located that concerns a hard non-functional requirement about the security of the system. On a lower level, a number of features are located that elaborate the use-cases and the non-functional security requirement. Usually, those features present close functionality, which can be realized in relatively independent units. For example, "Script Recognition" and "Image Binarization" are two features, the implementation of which involves clearly defined algorithms requiring exact type of input data and producing concrete results. Some of the features on the second level show design decisions, which are made in order to reach design objectives. For example, the "Protected Software Licenses" and the "Leveled User Authentication" are concrete decisions, which are made in order to improve the system's security.

10.7 Conclusion

In this chapter was shown that feature modeling is a technology with high potential; it can be used to support software evolution. This potential is due to the facts that feature models are capable of presenting domain concepts in a structured way and features are presenting abstractions, which are equally close both to system requirements and architecture. Hence, feature models are capable of bridging the abstraction gap between requirements and architecture.

In order to develop the potential of feature modeling of supporting software evolution, the technology needs to get an extended role in the software engineering process. This role includes use of feature models not only for requirements analysis concerning the design objectives of software system, but also for documenting specific design decisions, which are made to reach the objectives. In that context, feature modeling is a technology that has to be used both by system analysis and system design. As a result, feature models are capable of making a comprehensive overall system presentation.

The extended role of feature models in the software engineering process assumes relations between features and existing elements. Features need to be traced to requirements and to elements from the design space as well. Although these relations can be with unrestricted multiplicities, for the sake of simplicity of the approach and in order to introduce development discipline, the relations of features to other elements of the software engineering should be restricted as follows:

- Requirements to features as n to 1 relation;
- Features to design elements as n to 1 relation.

The ability of features to present design objectives also allows tracing between features and use cases. They complement each other and, as a result, perform a comprehensive overview of the statics and the dynamics within a software system. Similar tracing can be performed between features and non-functional requirements. Thus, a certain structure of feature models capable to support software evolution can be defined. Two levels of features can be distinguished, namely features related to use cases and/or to non-functional requirements and features detailing use cases and/or non-functional requirements. Preserving this levelled structure by building a feature model allows easier navigation within it and clearly shows the objectives and the design concerns in a software system.

Chapter 11

Feature-Assisted Reverse Engineering

11.1 Motivation

Previously in this thesis, it was mentioned several times that feature models can be eligible means of supporting software evolution tasks concerning reverse engineering. In this chapter, more attention is paid to this topic. The work described in the chapter shows how comprehensive and structured domain knowledge presented in a feature model can be utilized in program comprehension and architecture recovery activities.

At the beginning of the chapter, a way of building a feature model for an existing system is described. Later, on the basis of the established model, a method of program comprehension and recovering of architecture information is presented. The approach elaborates the idea of applying problem domain knowledge to reverse engineering tasks. A new viewpoint is added to the existing technologies described in chapters 7 and 8 in regard to the application of high-level design information in program comprehension and architecture recovery. The presented method uses the feature model structure and the requirements-features-architecture relations described in chapter 10. Due to the role played by feature modeling in the method, it is called **"Feature-Assisted Reverse Engineering"**.

To illustrate the presented method, examples are used from the studied image processing system.

11.2 Establishment of Feature Model for an Existing System

Earlier in chapter 5, while discussing domain engineering, feature models were presented as a result of domain analysis activities. Domain engineering and related techniques have been applied more widely in software engineering during the last decade, especially with the introduction of the product lines approaches. Nevertheless, a large number of software systems have been developed earlier, without using domain engineering techniques. Many of those systems are still subject to evolution and require reverse engineering. Therefore, in order to apply the ideas for supporting evolution tasks with feature models in the context of such systems, it is necessary to build their feature models initially.

In order to establish a feature model of an existing system, a number of activities have to be performed. First, information sources have to be collected and examined with the help of domain analysis methods (see chapter 5). On the basis of the elicited information, design objectives can be modeled. As a second step, the established model can be further developed by adding design decisions, which are extracted by studying design documentation and by architecture analysis. As a result, a feature model can be established keeping the hierarchical structure described in section 10.6. Parallel with the feature model, use cases for modeling system dynamics may also be collected. The establishment process is graphically presented on figure 11.1¹ and the different activities are discussed in detail in the next sections.

For example, in order to establish a feature model of the studied image processing system all documents describing requirements and system design have been collected and studied. On the basis of the acquired from the requirements documents information, a feature model presenting the design

¹On figure 11.1 are used UML [UML] activity diagrams notations.

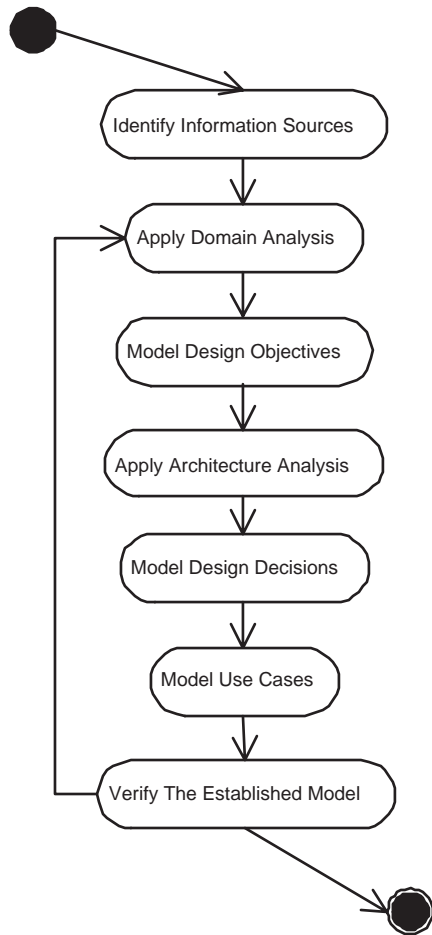


Figure 11.1: *Establishing Feature Model of an Existing System*

objectives of the system has been established. This model has been extended with the design objectives of the system, which have been elicited by studying the system design documentation and by analyzing the system architecture with the help of conventional reverse engineering methods (see chapter 7). Finally, the correctness of the results has been verified in a set of interviews with system experts.

11.2.1 Collection of Information Sources

There are various information sources that can be used to build a feature model of an existing system. Such are:

- system documentation including requirements and design specifications, user manuals and product descriptions;
- source code and source code documentation;
- interviews with system experts and users.

Usually, not all information sources are available in every system and furthermore, the reliability of the information differs greatly. As a rule, the most reliable information is contained within the source code and the source code documentation. Unfortunately, the complexity of this information is normally too high to allow identification of system features. For this purpose system documentation and results of interviews with system experts and users are more appropriate. Later, the identified features can be proved with the help of source code and related documentation.

11.2.2 Model Design Objectives

After the proper information sources have been identified, the design objectives of a software system can be modeled. For this purpose, both system and domain requirements have to be examined. At this step, the usual activities of domain analysis and requirements analysis can be applied:

- study of documents, forms, and guidelines;
- interviews with system users;
- interviews with experts for user support and system maintenance.

The activities listed above result in a feature model containing design objectives. In this model, the distinction between mandatory and optional feature has not been utilized yet. This is due to the reason that only existing features have been identified and no concept for variability analysis has been applied. Nevertheless, the model in this state is quite suitable for presenting system design objectives and thus for structuring domain knowledge needed in further reverse engineering tasks.

In order to observe the structure described in section 10.6, on the top of the hierarchical structure are positioned features corresponding to basic use cases and to non-functional requirements. Features detailing the above groups are positioned in lower levels.

11.2.3 Model Design Decisions

The feature model containing design objectives can be complemented with their corresponding design decisions. This can be done by eliciting design decisions with performing architecture analysis of the studied system.

The activity starts with studying existing documents about design and system architecture. The documents are studied using requirements derived from the previous step for comparison purposes. Knowledge about reference architectures of the domain and about design patterns should be included in the analysis process. In case such knowledge is not available, it should be obtained from interviews with system experts. The step results in a feature model presenting full set of system features, including design objectives and the corresponding decisions.

11.2.4 Collect Use Cases

In parallel to the activities aimed at the establishment of feature model for an existing system, use cases for the system can be additionally collected. The use cases information is obtained anyway while modeling design objectives. This information can be saved and utilized later for assisting the analyses of system dynamics.

11.2.5 Verifying the Feature Model

Both activities about modeling design objectives and design decisions can be iteratively repeated. After each iteration, the resulting feature model is refined according to the remarks of the system experts (users, developers and problem domain experts). In the verification, the following points should be considered:

1. correctness of the used terminology;
2. completeness of the model;
3. correctness of the hierarchical structure of the model.

There can be contradictions between the inputs of the different experts. For example, there can be discrepancies in the terminology used by system users, developers and domain experts. Often, the problem of different experts having a different understanding of the relations between system features occurs. That is the reason why there is a need of considering priorities for the different experts input, depending on the type of the verified features. For design objectives, input of domain experts and users is prioritized, while for verification of design decisions the input of developers should be considered more important.

The verifications assure the validity of the established feature model. Due to the considered expert opinions, it is possible to claim that the model presents comprehensive and valid information about an existing software system. On the other hand, taking the opinion of various experts into account also increases the clarity of the feature model and its value as a means eligible to communicate between different stakeholders in the software engineering process. Due to these special features, feature models can assist program comprehension and architecture recovery activities and, as described later in chapter 12, architecture restructuring.

11.3 Using Feature Models in Program Comprehension

As it was discussed in section 7.3, program understanding requires a model, which presents domain knowledge for a software system. The model should contain information about the domain boundaries, terminology and possible architectures. It also should be simple and comprehensive enough in order to be successfully utilized in an understanding process. In this section, it is shown that feature models can be eligible to solve these tasks. Furthermore, feature models may not only assist, but also guide a program understanding process.

Due to the well-structured presentation of problem domain knowledge, on one hand, and system design decisions on the other, feature models can be utilized as a tool guiding the understanding process for software systems. The structure of feature models described in section 10.6 provokes a top-down concept of comprehending the concerns in a software system. The described feature levels provide software engineers with abstract design information in the following sequence:

- a) At the top level information can be found about main system concerns including:
 - system functionality and
 - system non-functional requirements;
- b) At the second level within the hierarchical feature structure, information can be found about the decisions made to solve functional and non-functional system concerns.

The information provided by a feature model is the one needed by an experienced software engineer who is supposed to work on a barely known software system, which is the usual case for program understanding. The engineers are capable of complementing their general software knowledge with the problem domain one. Due to the comprehensive process of building a feature model for an existing system, the provided knowledge can be considered fully reliable.

The understanding concept provided from a feature model is a natural approach based on the principles of information decomposition. Following the nodes in a feature model top-down, reverse engineers can first get abstract information about the general functional and non-functional requirements in a software system and second, find out about the decisions made to satisfy those requirements. Sequentially, reverse engineers can build hypotheses about the architecture and source code needed for implementing system features. Thus, engineers can be released from the task of comprehending complex source code and system documentation and they can use them only as means of verifying their hypotheses. This way of work significantly reduces the complexity of program understanding. It combines a top-down hypotheses establishment process with bottom-up verification.

11.4 Feature-Based Architecture Recovery

In this section, the feature-based program understanding approach is further developed into a method (Feature-Based Architecture Recovery) of software architecture recovery. The method presents an incremental recovery process performed by establishment and verification of hypotheses. It elaborates the hypothesis establishment ideas presented in [Brooks et al. 1983], [Letovsky et al. 1986] and other reverse engineering approaches (see chapter 7). A hypothesis by the feature-based architecture recovery describes a supposed relationship between a feature and an architectural element.

The Feature-Based Architecture Recovery method is described in detail in the next sections, where attention is paid to the architecture recovery process and the means of verifying the recovery results.

11.4.1 The Architecture Recovery Process

The complete feature-based architecture recovery process is presented in figure 11.2. In general, the process is separated into three main parts, namely:

- A) establishment of feature model;
- B) generation and verification of architecture hypotheses for static architecture description;
- C) generation and verification of hypotheses for dynamic architecture description.

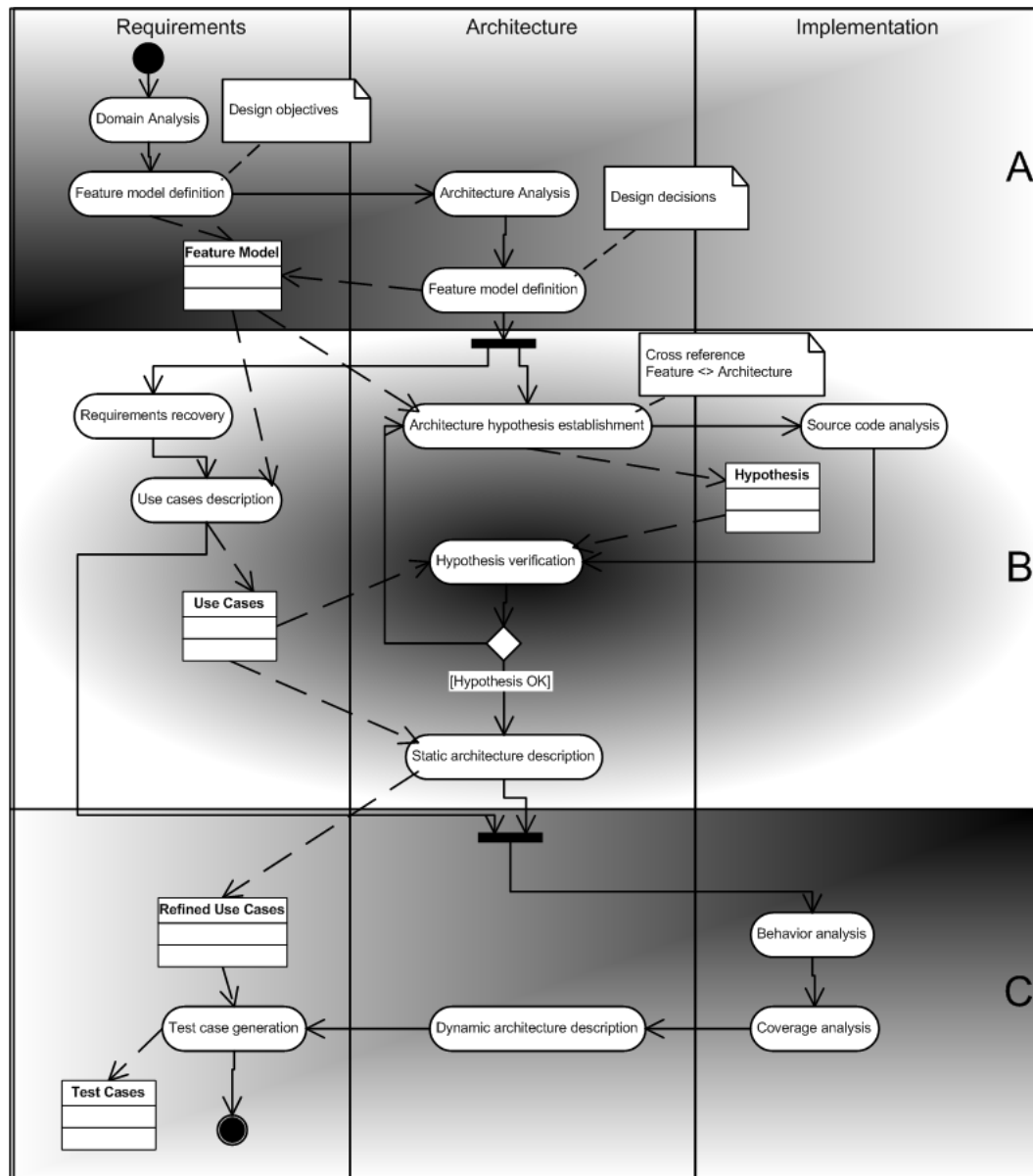


Figure 11.2: *Feature-Assisted Architecture Recovery Process*

The first part of the method activities (establishment of feature model) was already discussed in detail in section 11.2 and it is not necessary to pay more attention to it. The activities on generation and verification of hypotheses for static and dynamic architecture structures are of a greater interest here.

Generation and verification of hypotheses for static architecture description

Following the nodes in a feature model, reverse engineers can make hypotheses about relations between existing architecture and source code elements that realize the features in the software system. For example, on figure 11.3 is presented an extraction of a feature model describing the studied image processing system. The extraction is about a part of the system features concerning image storage capabilities. Naturally, one can assume that since these system features can be described in a separate branch of the feature model, those features are also realized separately in the system implementation and design. In that order, there should be a component or a subsystem dealing with image storage. As a confirmation, there really was a similar component in the studied system.

Further following the nodes within the sample model on figure 11.3, it is possible to decompose the image storage component. For example, if one takes the feature named "Recover images from previous run", which is a design decision caused by the "Reliability" feature; one should expect for it functionality allowing recovery of records to be implemented. This causes hypotheses about implemented logging, mirroring etc.

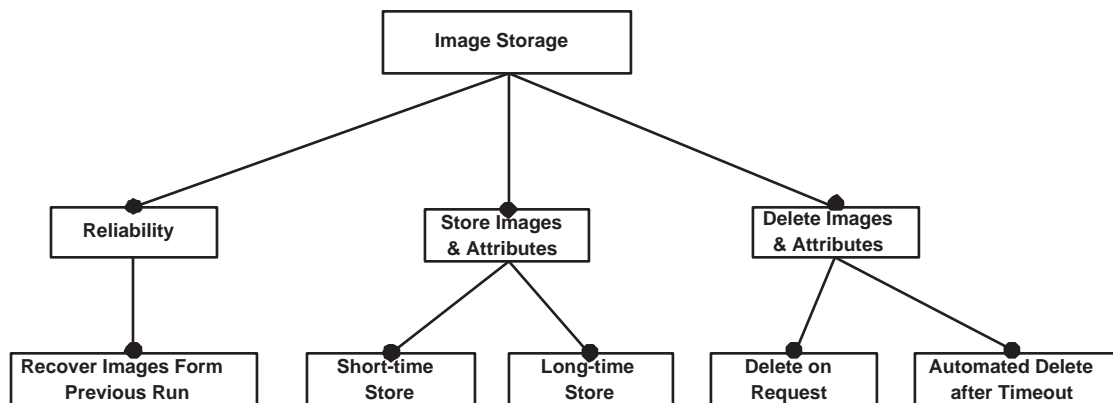


Figure 11.3: *Image Storage Capabilities - Extraction from Studied Image Processing System Feature Model*

The feature model itself is not sufficient to generate a full list of hypothesis about existing architecture elements. It can only support this process. For a comprehensive approach, the help of conventional reverse engineering methods like analyzing data structures, functional structures and control flow between modules is required. Nevertheless, the feature model guides this process and, as mentioned, complements the conventional methods with the provided abstract design information.

As a second step, the generated hypotheses have to be verified and the existence of relations between them and system features has to be proved. For this purpose, the established architecture hypotheses and the corresponding features can be organized in cross-reference tables (Tables 11.1 and 11.2). In brief, these tables show the relations between features, architecture hypotheses and parts of the source code connecting them. The verification procedure is discussed later in more detail.

Since the hypotheses establishment-verification activities are performed iteratively, the results of each iteration can lead to refinement of the hypotheses.

Generation and verification of hypotheses for dynamic architecture description

Similar to the activities on generation and verification of static architecture description, analysis of the architecture dynamics can be performed. Performing such analysis is also facilitated by the fact that use cases are collected along with the establishment of a feature model of the system. Furthermore, as described in section 10.6, by presenting the structure of the model, the use cases are linked with features from the model. Due to this fact it is possible, alongside the analyses of static architecture structures, to perform analyses of the dynamics of these structures.

As shown on figure 11.2, after finishing the static architecture recovery activities, use cases can be executed and coverage and behaviour analyses performed. The results of these analyses will reveal the dynamic relations between static architecture elements. In addition, it is possible to generate a set of test cases, describing the studied system.

11.4.2 Verifying Architecture Hypotheses With Feature-Architecture Elements Cross-References

To support the establishment and verification of traces between features and architecture elements, the feature-based architecture recovery method proposes usage of cross-reference tables. These tables show which feature corresponds to the respective hypotheses and vice versa. The usage of such tables also closes the hypotheses generation verification cycle (figure 11.4). Table 11.1 presents an example of a cross-reference table, corresponding to the feature model extraction from figure 11.3.

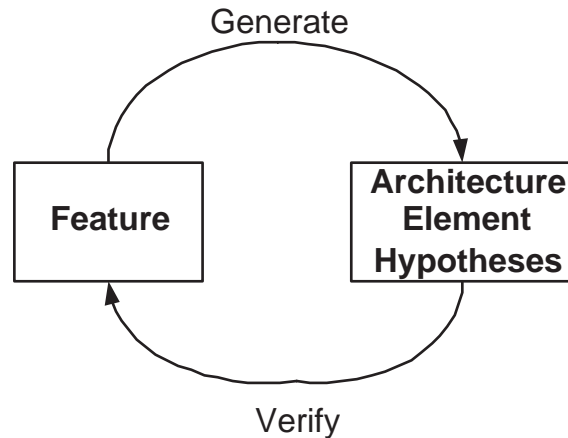


Figure 11.4: *Tracing Features to Architecture Elements*

Feature/Architecture Element	Image Container	Action Log	Container Mirror	Timer
Recover Images Form Previous Run	-	+	-	-
Store Images & Attributes	+	+	-	+
Short-time Store	+	+	-	-
Long-time Store	+	+	-	+
Delete Images & Attributes	-	-	-	-
Delete on Request	-	-	-	-
Automated Delete after Timeout	-	-	-	-

Table 11.1: *Features to Architecture Elements Cross-Reference Table*

The rows of a features-architecture elements cross-reference table contain the different features, while the columns contain the established architecture hypotheses. In the crossed cells, there are confirmation marks "+" for existing relations or disclaim marks "-" for non-existing relations between the listed elements.

The results of the established cross-reference tables are crucial to the verification of architecture elements hypotheses. If there is no feature corresponding to a hypothesis then the hypothesis is obviously wrong. In case there is a feature without a corresponding architecture element, a hypothesis is apparently missing or the tasks of some architecture elements have not been completely discovered. In any case, additional analysis is required. This is due to the fact that, according to the way the feature model is established, the model presents the actual implemented functionality and solved non-functional requirements. In that sense, it is impossible to have a feature without corresponding implementation.

The example presented in table 11.1 shows the results of cross-referencing seven features and four architecture element hypotheses. As it is visible from the table, no architecture hypotheses are available for three of the features connected with image deletion. Further analyses showed that in the case of the studied system the functionality of the "Image Container" and the "Action Log" components was not completely discovered. These components had implementation concerning image deletion as well. From the example, it is also visible that the "Container Mirror" architecture element hypothesis has no corresponding feature. The hypothesis was caused together with the "Action Log" one while searching for implementation of the "Recover Images Form Previous Run" feature. Obviously, the used image

recovery method in the studied system was based on action logging and the mirroring hypothesis was wrong.

11.4.3 Tracing Features to Source Code

The features-architecture hypotheses tracing is further developed by the feature-based architecture recovery method into features to source code tracing. This is done with the help of cross-reference tables similar to the ones described in the previous section.

The features to source code tracing aims at providing additional information in case the one provided by features-architecture elements tracing is not sufficient for verification of the established hypotheses. For this purpose, cross-reference tables showing relations between concrete source code elements and features (table 11.2) are built, for example relations between classes or functions and features.

Feature/Source Code Element	Class X	Function Y	Type Z
Feature 1	+	-	-
Feature 2	+	+	+
Feature 3	-	+	-
...	-	+	-
Feature N	-	-	-

Table 11.2: *Features to Source Code Cross-Reference Table*

Similar to the features-architecture elements cross-references, the rows of the features-source code cross-reference tables contain features, while the columns list the source code elements, for which the corresponding traces to features have to be verified. In the crossed cells existing relations are marked with "+" and non-existing ones with "-". For example, table 11.2 shows an existing relation between "Feature 1" and "Class X" and non-existing relations between the same feature and "Function Y" and "Type Z".

To identify the relations between features and source code elements, some of the other reverse engineering methods discussed in chapter 7 can be applied, for example [Eisenbarth et al. 2003]. It has to be mentioned that the features to source code cross-reference tables do not identify such relations; the purpose of these tables is only to structure them. Later, the results of both types of cross-reference tables (features-architecture and features-source code) can be compared and the required features-architecture elements traces verified. For example, the corresponding architecture hypotheses and the corresponding source code elements can be seen for the same feature. Further analyses can prove or decline relations between source code and architecture hypotheses, respectively verify or discard the hypotheses and the traces to features.

Features to source code tracing can become a complex task in many cases, especially for systems having huge complex source code and many features². For this reason, such traces should be considered only as an option and applied only in case the results of features to architecture elements tracing are not sufficient. That is why the example presented in table 11.2 is also an abstract one. For the studied image processing system, the verification of the established architecture elements hypotheses and the traces to features was performed only with features to architecture traces.

11.4.4 Using an Architecture Model as a Frame for Architecture Hypotheses

Earlier, in chapter 8, software architecture was presented as the design base for large complex systems. As it was mentioned, the main task of software architecture is to reduce the complexity of software systems. A number of methods for doing software architecture were discussed and two of them were pointed out as comprehensive and well-known approaches. These are namely: the "4+1 Views" model and the "4 View" model.

In order to perform architecture recovery, a proper model serving as a frame for the established hypotheses is needed. Both previously mentioned architecture models are suitable for common work with the feature-based architecture recovery method. Nevertheless, some peculiarities of the models should be taken into account.

²which is the usual case

It is best to use the "4+1 Views" model when systems with object-oriented source code are subject to reverse engineering; it is designed for developing such systems. The hypotheses established by architecture recovery can be directly placed in the logical view proposed by the "4+1 View" model. Later, this view can be complemented with architecture dynamics provided by behaviour and coverage analyses. Also, the use cases view of the "4+1 View" model serves well as frame for collected system use cases. Similarly to the logical view, the development and the process views of the "4+1 View" model can be frames for different architecture elements hypotheses.

The "4+1 Views" architecture model is not so appropriate for systems with a primary functional-procedural source code. For such systems, the usage of the "4 View" model can bring more benefits. The architecture recovery hypotheses can be collected in the conceptual view of the model and further refined in the module view. Finally, if required, traces to source code view can be also performed. Such examples are the traces from features to source code, presented in the previous section. Since the "4+1 View" model does not support use-case view, use cases collected by the architecture recovery should be considered in a separate model.

For example, for the studied image processing system the "4 View" model has been considered as more eligible to serve as a frame for the recovery of its architecture. Although the system is partially object-oriented implemented, the object-oriented concepts are not exclusively applied for its architecture. It was difficult to establish a logical view ³ of the system, but the conceptual view of the "4 View" model has fitted very well to the way the system architecture has been designed. Later, in chapter 14 a presented case study shows how the different architecture layers and components of the studied system have been put in the frame of the "4 View" model.

11.5 Conclusion

In this chapter an approach for feature-assisted reverse engineering was presented. The approach considered two reverse engineering tasks, namely: program understanding and architecture recovery. The approach elaborates the idea of applying domain knowledge in reverse engineering by introducing a new way to structure and present domain knowledge.

The establishment of a feature model for an existing software system is a key issue for the method presented in this chapter. A comprehensive procedure for collecting and modeling system features and use cases and further verifying the established model assures a solid base for performing feature-based program understanding and feature-based architecture recovery. The established feature model involves both design objectives and the corresponding design decisions.

Feature model of an existing software system was presented as means of assisting and guiding the program understanding process. The information provided by the feature model was recognized as a complement to the domain knowledge of software engineers. This includes information about system concerns, such as functional and non-functional requirements, and information about the decisions made to satisfy the requirements in a software system.

A feature-based architecture recovery method was presented as an approach supporting recovery of both static and dynamic aspects of software architecture. Similar to their use in program understanding, feature models were presented as means of guiding the architecture recovery process. In addition, feature models imply establishment of hypotheses for architecture structures and elements implementing system features. Cross-reference tables are used for verification of the hypotheses and the traces between features and architecture elements.

A significant advantage of the presented method of assisting reverse engineering tasks is that it states the basis for further methodological support of software evolution. As mentioned in chapter 10, feature modeling is the means of filling the abstraction gap between system requirements and architecture. Using the feature-based architecture recovery method is a good opportunity for performing system reverse engineering to state the basis for further application of feature modeling based techniques in software evolution. The results of the method including feature model and traces to architecture can be further utilized in analyzing system concerns, assisting architecture redesign, suppressing redundant development etc.

Unfortunately, it has to be mentioned that tool support for the presented method is not comprehensive. There are tools supporting feature modeling, which can help the establishment of the feature model required by the approach. Still, those tools are hard to integrate with existing software tools

³Proposed by the "4+1 View" model

and this fact complicated the application of the feature-assisted reverse engineering method. Also, many of the method activities have to be performed manually, including the building of cross-reference tables. Additional support from other reverse engineering approaches for establishment of architecture hypotheses and tracing features to source code needs to be considered.

Chapter 12

Feature-Assisted Architecture Redesign

12.1 Motivation

As it was made clear in chapters 4 and 8, the ability of software systems to evolve is determined to a great extent by the quality of their system architectures. In that sense, architecture redesign is a usual part of the software evolution process; it is required to keep the architecture close to the requirements of the problem domain.

Currently, there are many approaches supporting architecture redesign activities. Some of these approaches were discussed in chapter 7, including clone detection techniques, program transformation and refactoring. Nevertheless, as mentioned before, these approaches deal mainly with the source code of the software systems and pay less attention to higher-level design information, which is actually software architecture. In that sense, the support of these approaches for architecture redesign is not sufficient. This technology gap can be filled by a method based on utilizing the high-level design information provided by a feature model.

In this chapter, a method is presented called **”feature-assisted architecture redesign”**. The method is developed on the basis of the feature-assisted reverse engineering method described in the previous chapter but it does not necessarily depend on it. Among the requirements to this approach are the availability of feature model and traceability links between features and architecture components. These artefacts are utilized in this chapter for performing a number of analyses, the results of which can support software engineers in redesigning software architectures.

The feature-assisted architecture design method helps software engineers in performing four tasks, namely: detecting architecture disproportions and monoliths, detecting design redundancies, hinting separation of design concerns, and assuring architecture completeness after redesign. In addition, the method provides a technology for assuring compatibility between old and new architecture versions. All these tasks utilize techniques based on analyzing features and traceability links between features and architecture elements. These tasks are described in detail in the next sections together with an overall description of the activities executed by performing feature-assisted architecture redesign. Again, examples from the studied image processing system are used to illustrate the presented work.

12.2 Feature-Assisted Architecture Redesign - Process and Dataflow

The complete process describing the feature-assisted architecture redesign method activities is presented in figure 12.1¹. The method uses a number of analyses based on retrieving information from feature models and the traceability links between features and the architecture components constituting current system architecture. This is the reason why the existence of these artifacts is required and in case they are not present, their establishment has to be considered an activity of the method. A proper approach for establishing the required input is the feature-assisted reverse engineering method

¹For the diagrams on figure 12.1, the Gane-Sarson DFD-Notations [Gane et al. 1979] are used

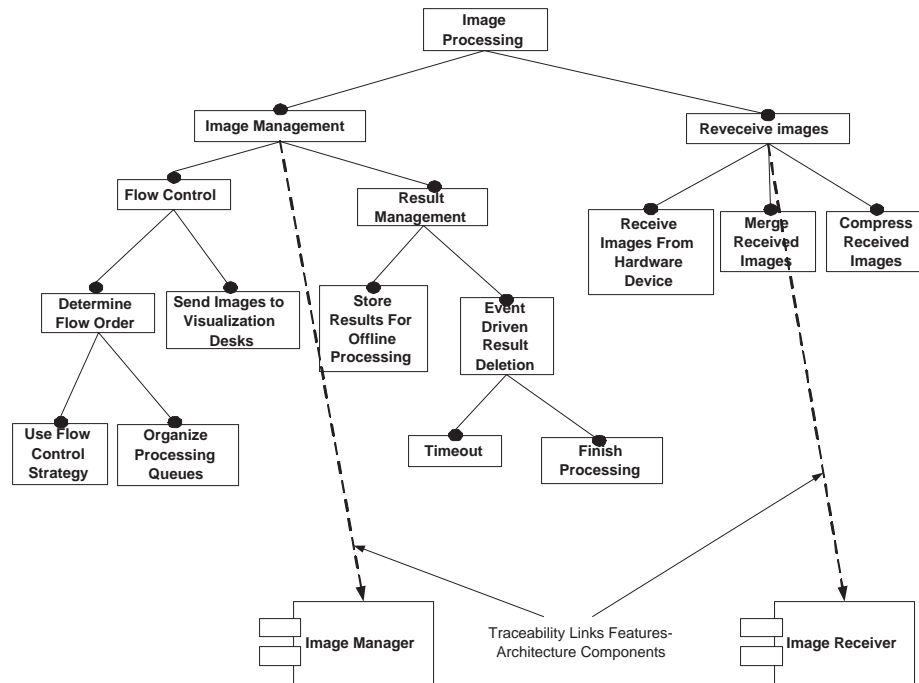


Figure 12.2: Sample Traceability Links between Features and Architecture Components

After assuring the proper input, the next activities supporting architecture redesign can be performed, namely:

- analyses of feature models and architecture traces;
 - detection of architectural disproportions;
 - clustering of features;
 - detection of redundancies;
 - hinting separation of system concerns;
- design of new system architecture;
- mapping of old to new system architecture;
- checking architecture compatibility.

The heart of the method comprises several analyses supporting software engineers in architecture redesign. These analyses detect possible disproportions between architecture components, reveal possible design redundancies and hint proper separation of system concerns. The results of these analyses can be used by software engineers for architecture redesign. It has to be noted that the new architecture design is primarily a manual process, for which, as described in chapter 9, certain professional capabilities, talent and experience are required. Nevertheless, the results of the analyses performed on the previous step are of good use and complement the experts' knowledge with facts that are often omitted and provide hints for making correct design decisions. The analyses supporting architecture redesign are described in detail in the next sections. Later, it is shown how the results from the method can be used to derive hints and clues useful for architecture redesign.

After architecture redesign is performed, the feature-assisted architecture redesign method provides a technology for assuring design completeness and compatibility with the old system architecture. This is the reason why a technology for mapping between old and new architecture components is provided. More attention to this technology is paid in section 12.6.

12.3 Detection of Architectural Disproportions

Evolution of software systems often leads to disproportional growing of parts, designed as equally complex in the initial system architectures. As a result, after a certain period of time the complexity

balance between different architecture components is completely lost and monoliths appear. It is known that monoliths have lower maintainability and hinder the ability of the software systems to evolve. Breaking the monoliths is a common task associated with performing architecture redesign.

Usually, detection of monoliths is not a difficult task since most of them are obvious and the related problems have been observed by system experts for a long time. Nevertheless, even the best experts sometimes miss important facts and, in a distributed development environment, the experts need arguments to support their opinions. This fact makes the need of a method for detection of architecture disproportions within software system architectures obvious.

Feature models designed in the way described in chapter 10 and built in the way described in chapter 11, are a good basis, on which analyses about existing architecture disproportions and monoliths can be performed. The analysis presented in this section examines system features collected in a feature model and the traceability links between features and architecture elements(components), in order to detect disproportions between the components. The method in brief:

- Calculates the number of features related to an architecture component.
- Calculates the average number of features implemented in all architecture components.
- Positions the architecture components according to the diversion on the number of related features towards the average number of implemented features.

All calculations are made on the basis of a feature model of the studied system and the traceability links to the architecture components. Due to the existence of traceability links, it is possible to count the number of features related to an architecture component. The average number of features is calculated as an average value of the number of all features and the number of all components.

Table 12.1 presents sample results of the method. The example is taken from the studied image processing system. The first column of the table shows the components composing the system architecture and the second column shows the number of features realized by each component.

Component	Features Per Component
IM (Image Manager)	53
FO (Flow Observer)	21
VD (Visualization Desk)	18
UII (User Interface Interceptor)	18
DISP (Dispatcher)	17
<i>OCR (Optical Character Recognizer)</i>	<i>16</i>
<i>ELMSGW (External Information Management System Gateway)</i>	<i>12</i>
<i>II (Image Injector)</i>	<i>10</i>
ISIF (Image Source Interface)	9
ICIF (Image Consumer Interface)	9
SC (Statistics Collector)	7
IDB (Image Database)	5
XVI (External Video Interface)	5
IR (Image Receiver)	4
ISDB (Image Streams Database)	3
TD (Test Desk)	3

Table 12.1: *Features per component classification example*

As shown in table 12.1, there is a significant difference in the number of related features between the components at the bottom and at the top of the table. The components at the top of the table, especially the IM (Image Manager) concentrate a lot of functionality and one can say that they have become monoliths. On the other hand, the components at the bottom of the table are quite simple although they are classified on the same architecture level as the other ones. The components in the middle of the table (in italic) implement a number of features close to the average for the system (13 features per component +/- 20% project specific tolerance) and can be considered well balanced.

12.4 Clustering of Features

Since features are described by a name, which is also composed of terms from the system vocabulary, it is possible to classify features in feature clusters. As it is shown in the next sections, the results from features clustering can be successfully utilized in a number of tasks concerning architecture redesign. In this section, a semi-automated method for clustering of features is described; in brief:

- Application of text analyses information retrieval techniques over a database of feature names in order to detect and order the main concerns expressed by the features. Two types of analysis are performed, namely:
 - words frequency analysis;
 - words categorization analysis.
- Semi-automated clustering of features according to the text analyses results;

In order to apply information retrieval algorithms, it is necessary to prepare a database with information that has to be analyzed. For this reason, the names of the features describing a feature model are put in such database, over which the information retrieval text analyses are performed. The analyses also need a thesaurus. In the studies described in this thesis, the thesaurus provided from the used information retrieval tool was used².

Since normally feature names are relatively short and the contained text information could be insufficient for information retrieval, requirements related to the features can also be added to the information database and analyzed. Nevertheless, even without taking the requirements into account one can argue about the correctness of the later described information retrieval analyses due to the following reasons:

- Feature names explicitly use domain technology, which concentrates significant domain information.
- Additionally, if feature models are created using the feature-assisted reverse engineering method, due to the verification procedure incorporated in the method, the correctness of the used terminology is most likely guaranteed (see chapter 11).

The first type of information retrieval analysis is the **words frequency analysis**. With its help, the most frequently used words within the studied database are discovered and classified. The result of the analysis is a report, which shows in how many records (in this case features) these words are mentioned and what percentage of the whole number of studied records they occupy. As a rule, the results contain only words that are met in a percent of the records greater than a specified value. Table 12.2 shows sample analysis results of the studied image processing system. The barrier occurs in at least 2.214% of the records. This barrier number was chosen as recommended by the used information retrieval tool. In general, the barrier occurrence is dynamically calculated and depends on the number of analyzed different words and the used thesaurus.

Word	Frequency
Image	20(21.74%)
Processing	17(18.48%)
statistics	6(6.522%)
device	6(6.522%)
control	6(6.522%)
result	5(5.435%)
resource	4(4.348%)

Table 12.2: *Words Frequency Analysis Example*

Another type of information retrieval analysis, which is performed before the beginning of the feature clustering process is the **words categorization analysis**. It aims at identifying categories and sub-categories of context-related words. The analysis searches for classes and subclasses of words depending on their relation in a word construction (for example, in a sentence or, in this case, in

²The information retrieval studies in this thesis were performed with the support of the PolyAnalyst [PA2003] tool.

a feature name). The result of the analysis is a categorization tree presenting the identified word categories and sub-categories. Table 12.3 shows a categorization tree example taken from the studied system. In the categorization tree, the number of records (features), in which the categorized word is met, is given in brackets.

image(20)	convert(4)
	compress(1)
	TIFF(1)
	source(2)
	format(2)
	others
...	
statistic(3)	report(3)
	counter(2)
	record(1)
...	

Table 12.3: Words Categorization Analysis Example - Categorization Tree

The results of the above described analyses are used as input to the **feature clustering process**. The aim of the process is to identify and group features describing similar or logically related functionality and to give a proper name to the corresponding group (feature cluster). The clustering process is semi-automated. It is manually driven from experts but a tool for supporting them has also been developed. The tool helps experts to navigate through features and define feature clusters. The assignment of features to different clusters is manually done by the experts. Figure 12.3 shows the activities and the dataflow of the features clustering process.

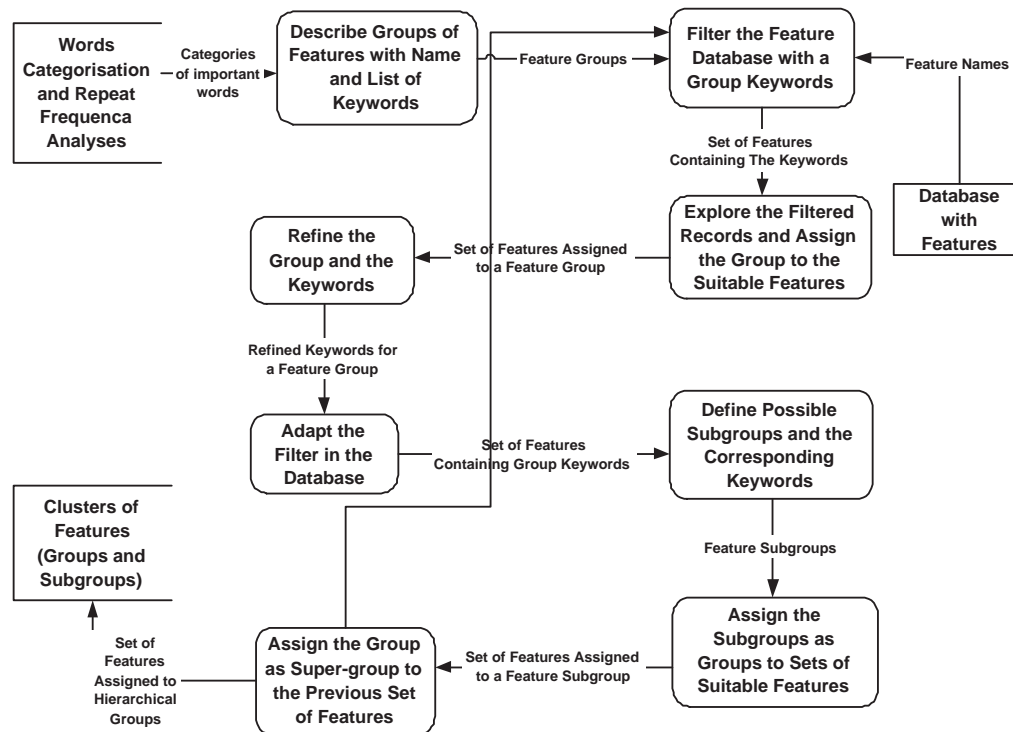


Figure 12.3: Features Clustering Process Activities and Dataflow

The results of the information retrieval analyses are used for definition of feature groups (clusters of features). Each feature group is given a unique name and list of keywords. The words from the word frequency analysis determine the names of the groups. The words from the word categorization tree determine the list of keywords for the groups. In addition, the experts can extend and tune the

list of keywords and group names. They can also define groups that have not been detected in the information retrieval analyses. Figure 12.4 shows an example form for editing feature groups.

ID	Group Name	Keywords
1	NoGroup	
2	File IO Operations	Load, Save, Store, Delete, File, Container, Erase, Disk
3	Image processing	Image, Cut, Rotate, Zoom, Region, Find, TIFF, Convert, Compress
4	Communication	Interface, Message, Transfer, Receive, Send, Communicate, Protocol, Cor
6	Statistics	Statistic, Counter, Report, Collect
7	Result management	Results, Attributes
8	Image IO operations	Image, Receipt, Retrieve, Store, Recover, Delete
10	Image	Image, Attribute, tiff, source, request
11	Diagnostics and testing	Monitor, Log, Diagnose, Trace, Debug, Test

Datensatz: 2 von 21

Figure 12.4: Features Clustering Tool - Feature Groups Edit Form

Once the feature groups are defined, features have to be assigned to them. For this purpose all features are listed in a feature group assignment form (figure 12.5). The experts iteratively filter the listed features. The applied filter is a logical union of keywords describing a feature group (e.g. "Load or Save or Store"). After the filter is applied the resulting records are explored and the experts assign the appropriate features to the group. The filter and the keywords can also be refined after the filtering results are considered. In case of refinement, the new filter has to be applied. If the new filtering results are satisfactory (the listed features belong to the same group), all appropriate features are assigned to the group. The filtering and assignment steps are repeated for each defined group.

	Architecture Component	Feature	Group	Super Group
1	Architecture Component			
2	VD (Video Desk)	Control the user input	User Control	User Services
3	VD (Video Desk)	Image processing	Image processing	Image
4	VD (Video Desk)	Provide graphical interface to user for the manual processing of the images	Image coding	User Services
5	Dispatcher	Delete messages when no client requires them	Communication	NoGroup
6	Dispatcher	Provide status information to and about the connected clients	System monitoring and control	System Services
7	Dispatcher	Dynamically determine the receiver of a particular message	Communication	NoGroup
8	Image Store	Per machine statistics	Statistics	Statistics
9	Image Store	Store images and attributes	Image IO operations	Image
10	Image Store	Overall Statistics	Statistics	Statistics
11	Image Source Interface	Diagnostic commands - set up the image source	Hardware interfacing	NoGroup
12	Image Source Interface	Connect an image source control interface to the system	Interfacing	NoGroup

Datensatz: 119 von 284

Figure 12.5: Features Clustering Tool - Feature Groups Assignment Form

For better grouping of features, it can be necessary to break some groups into subgroups. The subgroups are defined as normal groups and considered as such in the feature groups assignment form.

The difference is that the original group is considered as a super group in respect of these features. This technique allows reaching an unlimited hierarchy of feature groups. When the grouping is finished the resulting feature groups will present clusters of features, which have the same concern.

Table 12.4 shows some sample groups, which have been identified in the studied system. Table 12.5 lists a set of sample features, which have been classified to "Image TIFF Operations" and "Image IO Operations" groups. Figure 12.6 is an extraction of a report showing feature-grouping results (established feature clusters).

Feature Group	Keywords
Input Output	Load, Save, Store, Delete, File, Container, Erase, Disk
Image TIFF operations	Image, TIFF, Convert, Compress, Store
Image IO operations	Image, Receipt, Retrieve, Store, Recover, Delete
Statistics	Statistic, Counter, Report

Table 12.4: *Sample Feature Groups and The Corresponding Keywords*

Comp.	Feature	Group	Super Group
II	Convert raw image data into standard format (TIFF)	Image TIFF operations	Image
ISIF	Compress images (TIFF FAX4/JPEG)	Image TIFF operations	Image
IR	Compress received images (TIFF FAX4)	Image TIFF operations	Image
IDB	Store images in TIFF format	Image TIFF operations	Image
IS	Delete images	Image IO operations	Image

Table 12.5: *Features Classified to "Image TIFF Operations" and "Image IO Operations" Groups*

Super Group	Group	Component	Feature
Image	Image IO operations	IS (Image Store)	Retrieve images and attributes
			Store images and attributes
			TD Store loaded images into IS container style
	Image TIFF operations	IDB	Store images in TIFF format
		II	Convert raw image data into standard format (TIFF)
		IR	Compress received images (TIFF FAX4)
		ISIF	Compress images (TIFF FAX4/JPEG)

Figure 12.6: *Features Clustering Tool - Groups-Features Report (Established Feature Clusters)*

The established feature groups (clusters) from the clustering process are further utilized in the feature-assisted architecture redesign method for detection of redundancies within the system architecture and for hinting the separation of system concerns.

12.5 Detection of Redundancies

The redundant development is a common problem for long-life software systems. As stated in the beginning of this thesis, in many cases the evolution changes within these systems are performed in the environment of limited development time, which often causes usage of short-term solutions. It is known that a lot of code is developed using "copy paste" techniques and there exists a wealth of literature in approaches that deal with this problem, known as clone detection. In chapter 7 of this thesis, attention was paid to these techniques also. Unfortunately, "copy paste" development is not the only reason leading to redundant development. Many problems come from the system design itself. Due to the same problems supporting evolution changes, a lot of requirements are solved more than once. Implementation for the same feature can be found more than once in different architecture components. Sometimes, the similar features have slightly or even much different names in the different architecture components but a more detailed expert evaluation can show that those features actually realize the same system requirement.

Since the clone detection techniques work on source code level and omit high level design information, they are difficult to apply for detecting redundancies caused by architecture design problems. Feature-based methodology seems to be much more promising for solving such problems. As defined in chapter 10, a feature model stands between architecture and system requirements and considers them both. It shows the system concerns from one side³ and the importance of these concerns for the system architecture from another one. The feature model is also traced to the architecture components implementing those concerns. In that sense, analyses of features and their traceability links⁴ to architecture components can show redundancies caused by system design. In case a duplicated design objective or decision is detected within the feature model it can be shown if this concern is redundantly realized by the system architecture as well.

Figure 12.7 shows an example of possible redundancies detected in the studied image processing system. A set of five features (left) has been implemented in different system components (right), which finally have performed the same tasks (top). All of them have performed protocol mapping from external protocols to an internal one, but there have been several different implementations of the mapping functionality and no unified interface to the native protocol, although it was the same for all components. Due to the redundant design, the studied system has suffered performance problems, since the different components have been running as separate processes, which is proved to be more resource-consuming than multi-threaded architecture for example.

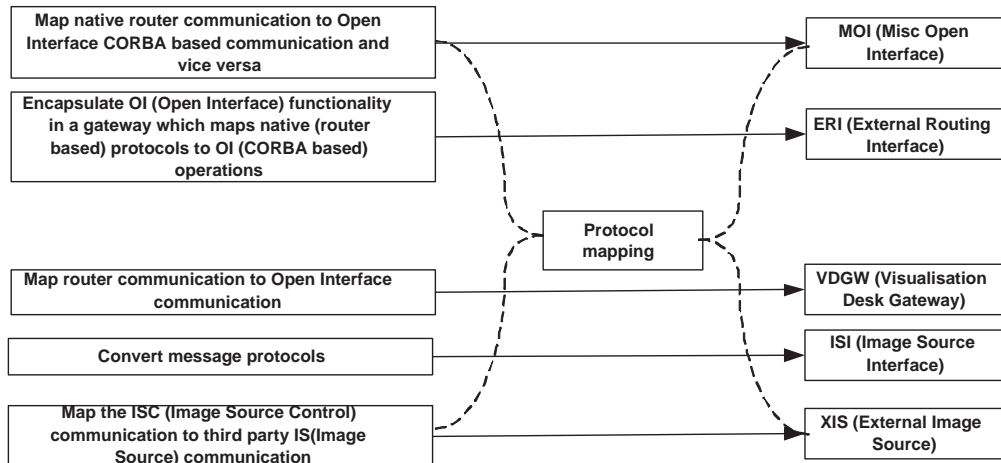


Figure 12.7: Possible Redundancies Detected in Studied Image Processing System

The results of the feature clustering together with the traceability links between features and architecture components can be utilized in a technique for detection of design redundancies. The features classified in the same cluster can be **examined for similarities**. In case similar features are found, the traceability links will show in which architecture components these features are implemented,

³As defined in chapter 10, a feature model contains both design objectives and design decisions within a software system.

⁴Both are presented by a feature model.

which will point to the redundant design. In order to support redundancies detection, the tool used for feature clustering has been extended with a new form helping the similarities examination. Figure 12.8 presents a snapshot of this form. The form also allows definition of new features merging other similar features. Table 12.6 shows an example of features from the "Image TIFF Operations" group, which have been considered similar and also the resulting new feature. Three of the listed four features had the same tasks and were considered identical.

Comp.	Feature	Resulting Feature
ISIF	Compress images (TIFF FAX4/JPEG)	Compress Images In TIFF format
IR	Compress received images (TIFF FAX4)	Compress Images In TIFF format

Table 12.6: Similar Features From The "Image TIFF Operations" Group

Figure 12.8: Features Clustering Tool - Similarities Examination Form

12.6 Hinting Separation of System Concerns

In chapter 10 during the description of the structure of the feature models used in this thesis it was noticed that a feature model considers both design objectives and design decisions of a software system. In addition, a relation to system use cases is also taken into account in the feature models. All these facts show that the information contained in a feature model is eligible to present the concerns in a

software system. In that sense, the information obtained from clustering features hints the separation of concerns within a software system.

The clusters of features can be used as a frame for determining the required components and interfaces in a re-designed system architecture. For example, in table 12.5, an example was given with two identified clusters of features, namely: "Image TIFF operations" and "Image IO operations". Both clusters belong to the "Image" supercluster. On the basis of these results, an "Image" component can be defined, which supports two interfaces corresponding to the two different clusters. This component is illustrated on figure 12.9.

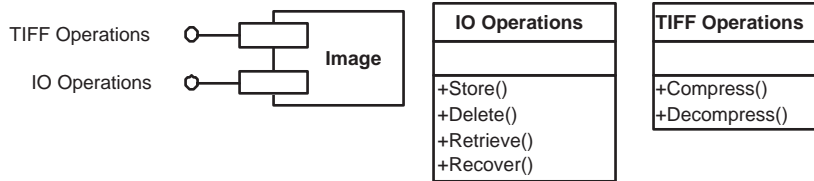


Figure 12.9: *Hinting Separation of Concerns - Image Component Example*

Architecture components determined on the basis of the hints provided by the features clustering simplify to a great extent the system architecture and remove the design redundancies. Figure 12.10 illustrates this statement. On the figure, both the determined new sample "Image" component (right) and the components from the old architecture (left) of the studied image processing system are shown in parallel; they have implemented the same features (middle). The difference between the two solutions is obvious.

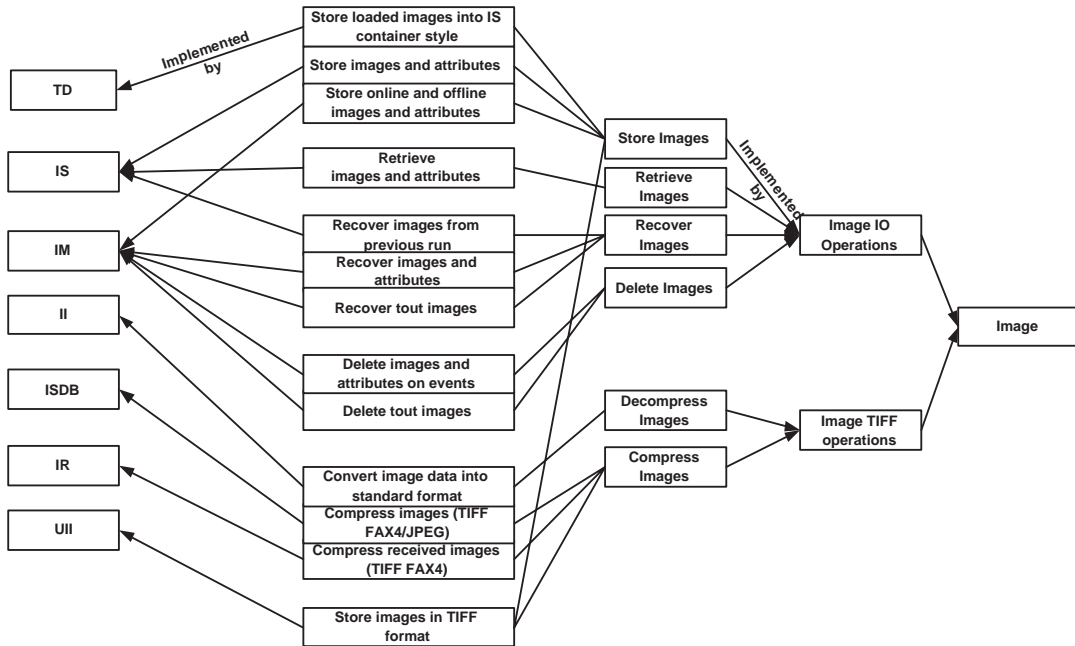


Figure 12.10: *Newly Established "Image" Component in Comparison to Components from the Old System Architecture of the Studied Image Processing System*

12.7 Deduction of Clues and Hints for Architecture Development

The results from the analyses described in the previous sections can be summarized in a set of clues and hints, which has to be taken into account by performing architecture redesign. Namely:

- The redesigned system architecture should consider and if necessary should correct the disproportions between the architecture components. In case of monoliths, those have to be broken

into finer grained components.

- The number of related and implemented features can be considered as a complexity metric for architecture components. Although such metric is very subjective and not an exhaustive one, it gives quick and good "first look" orientation for the complexity of architecture components. Keeping an eye on the number of features covered from the designed components helps avoiding monolithic development.
- The redundancies pointed by the redundancies analysis have to be removed in the redesigned system architecture.
- The established feature clusters can put the development on the right track finding the right architecture components. In general, the clustering results are not sufficient for complete components design, but the information provided by the feature clusters gives a good clue to the architecture designers. It defines in a certain extent the borders of the main system components and their interfaces.

12.8 Assuring Completeness and Appropriateness of Architecture Redesign

One of the most difficult tasks of architecture redesign is to assure that the redesigned system architecture is capable of meeting all requirements, which the older one fulfilled. Especially in cases of redesigning architectures of large and complex systems - the usual case in software evolution - it is absolutely necessary to apply a method that assures that all system features are considered by the redesigned architecture. Otherwise, the redesign risk is too high and the success of the approach is compromised. Due to the traceability links between architecture components and features, the feature-assisted architecture redesign method can provide such a technique assuring completeness and appropriateness of the architecture redesign.

The idea is simple. Features from the feature model used by architecture redesign can be traced both to new and old architecture components. This means that the feature model can be used as a map between the two architectures. Such mapping will help software engineers to follow the relations between the two architectures and to assure that the newly designed components meet the requirements met by the old ones. Furthermore, following the traces between features and new architecture components can show whether all features have respective implementing components in the new system architecture. Such a technology will immediately show possible inconsistencies within the new designed architecture.

An example of mapping old to new architecture components was already shown on figure 12.10. The presented diagram shows how the newly designed "Image" architecture component and its interfaces map to components from the old system architecture (TD, IS, IM, ...). As shown in the figure, the new component implements features that have been distributed over many components from the old system architecture. Without applying methodological tracing (mapping) between the two architectures, the risk of "forgetting" a feature while redesigning the architecture is undesirably high.

12.9 Conclusion

In this chapter, the feature-assisted architecture redesign method was presented, which is a methodological approach of supporting redesign of system architectures. The method has utilized information contained in feature model and traceability links between features and components.

The presented method shows how system experts can get good clues and hints about architecture redesign activities from problem domain information gathered in a feature model of a software system. The presented work demonstrated how analyses performed over feature models and application of information retrieval techniques can hint system experts of existing design problems, such as unbalanced architecture and design redundancies. In addition, the proposed analyses provide hints about overcoming the design problems and about possible separation of system tasks over architecture components. The presented work is also concerned with the problem of assuring completeness and appropriateness of architecture redesign and provides a methodological way of supporting this task based on mapping through features between old and new architecture components.

Similar to the feature-assisted reverse engineering method, the feature-assisted architecture redesign method is primarily manual. Although tool support is assured for some of the method activities, many of them require manual expert work. The field for automating the approach is still open, especially for the tasks concerning mapping between two architecture versions. During the research work, a prototype of such tool support has been developed but it remains only an experiment and future work on it is to be considered.

The feature-assisted architecture redesign method contributes to reaching the main aims of this thesis, namely preventing the need of software revolutions, increasing the overall lifetime of software systems and reducing maintenance costs. The systematic approach for architecture redesign provided by the method increases the effectiveness of the evolution changes and thus prevents their transformation into "small revolutions". The method assures a smooth and effective migration strategy between two versions of a system architecture, which significantly reduces the risk of performing revolution changes, i.e. the changes become more acceptable. This fact has a positive influence on the probability of postponing the changes, which means that the management will be more willing to allow the changes and the architecture of a software system will be able to follow timely the development of the problem domain. Hence, the software systems will be capable of satisfying the requirements of the problem domain and the overall lifetime of the software systems will increase. Finally, all these positive facts directly reflect on the software maintenance costs which they should reduce.

Chapter 13

Feature-Based Requirements Engineering for Supporting Software Evolution

13.1 Motivation

In the chapters 11 and 12, two methods were presented that deal with the technical issues of the methodological support for the software evolution process. Although they contribute reaching the aims of this thesis, a single technical solution is not sufficient. The process part of software engineering must also be taken into account in order to achieve high software evolvability.

What does the software evolution process usually look like today? Figure 13.1 tries to give an answer to this question. It visualizes the software evolution process, which results from following the state-of-the-art software engineering technologies discussed in part two of this thesis¹. As shown in figure, in most cases a number of deliverable end user systems are developed on the basis of the system architecture. Although they all have different requirements, some of which are not supported by the system architecture and require its evolution, the latter is not performed. There is no feedback from the deliverable systems into the system architecture. The way system architecture is usually evolved is by doing R&D (research and development) projects for this purpose. Since R&D projects usually have a large time span between them, there might be cases of a deliverable system developed on the basis of another one (figure 13.1, A). Such kind of development usually leads to decay of system architecture and, in many cases, it causes the R&D projects to turn into reengineering ones. This fact, which was pointed at the beginning of the thesis (see chapter 3), transforms the need of software evolution into a need of performing revolution within a software system.

Another common problem of the software evolution process today is the applied versioning strategy for system architectures. The usual architecture versioning strategy is illustrated on figure 13.2. As shown, there can be different versions of system architectures realizing different sets of features. For example, for the set of features {F3, F7, F8} a new version of the system architecture has been defined (version 1.1). Similarly, version X of the system architecture has been defined, when features F12, F13, F14 and F15 have been implemented. The different architecture versions are obtained after the subsequent R&D project. Such a versioning strategy usually causes many problems concerning backward compatibility of two different versions of system architectures and sometimes problems concerning the consistency of new architecture versions. As a rule, it is almost impossible to perform maintenance of deliverable systems based on previous architecture versions using the latest one. This causes maintenance costs to increase since maintenance for all architecture versions having deliverable systems in use is required.

In chapter 5 and later, in chapter 9, attention was paid to a number of domain engineering approaches and the product lines approach as means of improving the software engineering process by

¹Figure 13.1 refers especially to the software evolution process, which occurs if a product line development is applied to a software system (see chapter 9). The product line development has proven to be the most effective technology for developing highly evolvable software systems. The process presented on figure 13.1 shows the common industrial practice used for applying software product line development. As a particular example the studied image processing system has been used.

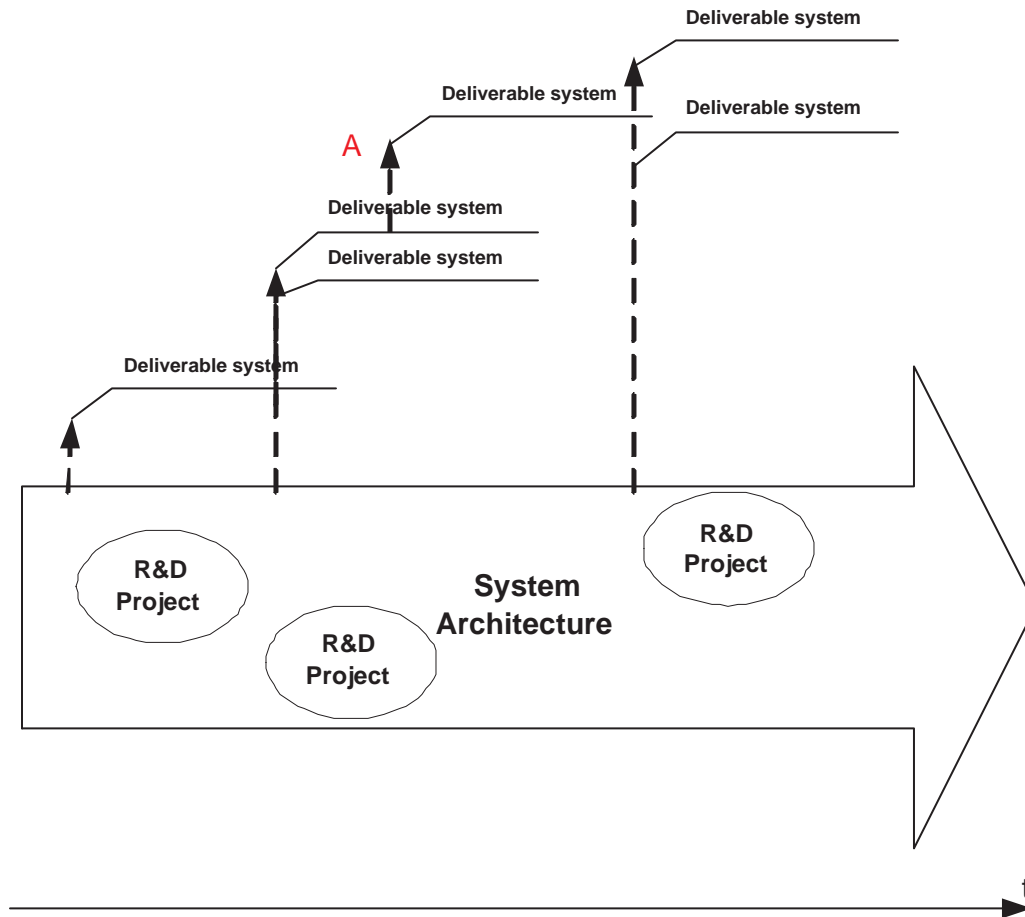


Figure 13.1: Usual Software Evolution Process by Applying Product Line Development

considering the variability of systems in a problem domain and by applying a strategy to deal with it. Although those approaches are a great step ahead, they still have gaps considering the software evolution problems. A methodological process for keeping system architectures up-to-date with problem domain changes is missing. In chapter 4, when discussing the life cycle models of software systems, the agile methods of software development were considered. It was pointed out that in general, agile methodologies are hard to apply for organizations developing long-life software systems but there are some principles of the agile development that can be taken into account. One of these principles is the application of early changes to the system architecture as soon as they are required.

This chapter introduces a methodology, which applies feature-modeling techniques in the requirements engineering process. It deals with the problem of outdated software system architectures. The approach aims at improving the communication between requirements engineering and architecture design processes, on one hand and at increasing the influence of requirements and their structure over the system architectures, on the other. Due to the extended role of feature modeling as defined in chapter 10, feature models are now used as conductors supporting reflection of requirements changes over the system architecture. During requirements engineering, timely changes are made to the structure of the feature model and force architectural changes. Thus, the reengineering risk is split in less risky small steps. The feature model is additionally used as means of synchronizing architecture changes required by parallel project development in order to avoid redundant design and development.

13.2 Prerequisites for Performing Feature-Based Requirements Engineering

The feature-based requirements engineering approach requires a set of prerequisites. If these prerequisites are not available, they have to be considered as initial activities of the approach. The best time

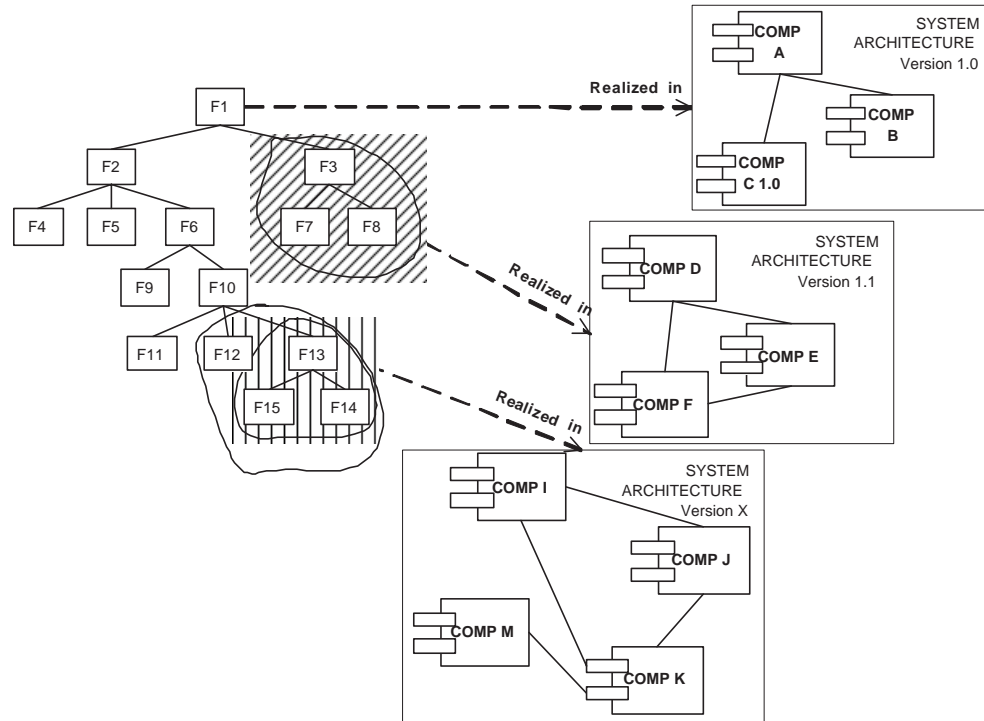


Figure 13.2: *Common Architecture Versioning Strategy*

to perform these activities is the time when new significant changes to the system architecture have to be made. As described in the previous section, this is done in cases when an R&D project is started. The beginning of an R&D project is also the best time to introduce the new requirements engineering approach in a software organization.

The prerequisites for applying feature-based requirements engineering are:

1. The software system has a feature model, which considers the structure described in section 10.6. The last requirement means that:
 - The hierarchy of features within the feature model presents the impact of the features over the system architecture.
 - Both, design objectives and design decisions are reflected within the feature model.
2. Traceability links between requirements and features are existent.

In case the required prerequisites are missing, the feature-assisted reverse engineering method is a good way of obtaining them. But similar to the statement made during presenting the feature-assisted architecture redesign method, other approaches of obtaining feature model and traceability links between features and architecture components of a software system can be used. In case of product lines development, these elements are probably already existent.

13.3 Feature-Based Requirements Engineering Rules

The feature-based requirements engineering method states a set of rules, which determine the frame of the approach. The rules are defined according to the evolution problems observed in the software systems, which according to the gained experience concern mainly weaknesses within the requirements engineering phase of the software engineering process. For this reason, the rules aim to improve the integrity of the requirements engineering process and the communication between the requirements engineering and the architecture development processes. A higher influence of requirements engineering over the system architecture is also taken into account.

The rules are namely:

1. The relations between system features² and components from the system architecture have to be strictly kept. The hierarchy of features within the feature model has to correspond strictly to the hierarchical relations of system architecture components. The restrictions of the features-components relations described in chapter 10 need also be kept in mind.
2. The features hierarchy has to be changed as soon as necessary. The feature model has to be kept up-to-date with latest system domain changes.
3. The feature model has to be centrally maintained and be visible from different system stakeholders. The management should be the owner of the model, the system architects should cover its maintenance and all developers have to be capable of viewing and using the feature model.
4. A distinction needs to be made between features with general importance for the system and those valuable only for a single customer deliverable system. Features without general system importance have to remain only within the scope of the specific deliverable system. Such features should not influence system architecture. The system architecture has to consider only the features with general importance for the whole system.
5. Features developed for a deliverable system, but considered as generally important have to initiate an update to the system architecture.
6. Requirements engineering for new system requirements has to be performed together with an estimation of the influence, which the new required features will have over the features hierarchy. The higher the level in the features hierarchy at which the new features have to be placed is, the bigger their influence over the system architecture (respectively their cost).
7. New type of features, namely pending features have to be introduced. These are the features developed within deliverable system projects and valuable as general system features but intended for implementation after a certain period of time.
8. Developers have to get a more important role in estimating the system features. They should be able to determine system features as well. The management should decide mainly for the importance of the features.
9. The versioning strategy for the system architecture has to be changed from complete architecture versioning into versioning of single architecture components.

The first two rules assure that changes of the feature model will be followed by changes within the system architecture. Thus, if the feature model presents the actual requirements of a problem domain, the system architecture will be able to meet them. Those rules actually realize the idea of the agile methods of applying early refactoring within a software system. The feature model is always kept up-to-date with the problem domain, i.e. continuously changed. As long as there is a change within the feature model of the system, this change will force an update of the system architecture when rule number one is kept.

The third rule realizes the idea of improving communication between the stakeholders. The central role of the feature model makes it a common means of communication, which allows different stakeholders to share domain knowledge.

The fourth rule assures stability of the system architecture. The rule helps to avoid aggravating the architecture by making it compliant with requirements important only for a single deliverable system but with no general importance. The system architecture only has to meet requirements shared between many systems based on it. Otherwise, its evolution and respectively the evolution of the whole system will be hindered by implementing less important and sometimes contradicting requirements, which finally remain only in the scope of a single deliverable system.

The fifth rule helps avoiding redundant work and implementing a feature more than once. The problem of redundant implementation occurs in many systems. In chapter 12, a technical solution to deal with it was presented, but it is better to avoid the problem at organizational level. As it was shown on figure 13.1, there is usually no relevant feedback from deliverable systems into systems architecture. The information flow is only from the system architecture to the deliverable systems. If the system architecture has to be extended with a feature developed in a deliverable system usually

²Features from the feature model of the system

this is performed at the next R&D project. This causes redundant development. In case a new feature has to be reused in several deliverable systems, they have to wait until an R&D project is finished or to develop the feature for themselves, which is the usual case because of the normally restricted project schedules. Obviously, such a way of working causes dissipation of development efforts and resources. Thus deliverable systems have to be able to feedback important updates to the system architecture.

The sixth rule supports the correct estimation of costs of changes and, respectively, better time planning. Hence it is possible to avoid time pressure problems. The latter are in many cases the reason for bad development.

The seventh rule introduces means of synchronizing parallel work done for different deliverable systems or R&D projects. Using pending features is a way of marking a feature within the feature model, which is planned for development within an already executed project. The pending features are not implemented within the system architecture but the exact time when they will be is known. Using such synchronization, again helps avoiding dissipation of development resources and redundant work. It allows a new development project to use the future benefits to be provided by an already executed one.

In many cases, developers are the best experts for a solution within a system domain. Developers are aware of how certain requirements can be met by the system features. In that sense, letting developers influence the system features and respectively the feature model is a reasonable idea. This statement is realized by rule number eight.

The ninth rule deals with the problem of architecture versioning strategy. It was discussed at the beginning of this chapter that applying versioning for the complete system architecture causes maintenance, respectively evolution problems. Multiple product versions lead to increases in maintenance costs. Component versioning seems to be more appropriate in the context of long-life and highly evolvable software systems. Later in section 13.5.2, detailed attention to this topic is paid.

13.4 Feature-Based Requirements Engineering Process

Application of the rules defined in the previous chapter leads to an extended, software-evolution-oriented requirements engineering process. The process is described in detail in the current section; attention is paid to the activities and the dataflow between the actors taking part in it.

13.4.1 Feature-Based Requirements Engineering Actors

There are three main actors taking part in the requirements engineering process, namely product managers, system architects and developers. Normally, those stakeholders are present in all companies developing long-life software systems. Usually, only the management and the system architects are involved in requirements engineering. Developers are only consumers of the process results and their expert knowledge is neglected. The feature-based requirements engineering corrects this problem. The responsibilities of the stakeholders in the process are as following:

- Managers define system requirements and decide on the importance of system features.
- System architects design the system architecture, specify system requirements and distribute them to developers.
- Developers determine the necessary features that can meet the system requirements, consider them with the system architects and finally implement them.

Such requirements engineering roles of the stakeholders allow better consolidation of the knowledge possessed by a software development organization. This fact is illustrated with the requirements engineering process activities and dataflow, which are presented in detail in the next sections.

13.4.2 Feature-Based Requirements Engineering Activities

The activities performed by the different actors within the feature-based requirements engineering process are shown on figure 13.3³. The figure is divided in three parts concerning the activities of the different actors.

³For the diagram of figure 13.3 the UML notations are used [UML].

The requirements engineering process starts with eliciting and defining new requirements for a deliverable system from the management. System architects perform mapping between those requirements and existing system features and determine if all of the requirements can be satisfied by the current system architecture. Checking for eventually required pending features is also performed. In case all requirements for the deliverable system can be met by current system features or, additionally by pending features, the requirements engineering process ends with a positive result. No new development and changes to the system architecture are required.

In case the requirements for a new deliverable system cannot be met by either current or pending system features, determining new features is required. The system architects determine if the new features will cause changes to the system architecture. If the architecture needs to be changed, then the importance of the new requirements is evaluated by system architects together with the management. In case the new requirements are considered important as general system requirements, changes to the system architecture and the system feature model are initiated. At that point, only changes to the hierarchical relations of the feature model are allowed, without adding new features. The system architecture changes concern determination of new architecture components and redistribution of system concerns between the existing components. The hierarchy of the feature model is updated in order to correspond to the architecture changes. If the new requirements are not considered to be of general importance for the system, they remain only as specific ones for the corresponding deliverable system and do not cause changes of the feature model hierarchy and consequent system architecture changes.

After the importance of new requirements is evaluated and the system architecture changes are defined, system architects divide the requirements into packets, which are forwarded to different groups of developers who are considered experts in special types of requirements. The developers process the requirements assigned to them and determine what new features need to be implemented in order to meet the requirements. Estimations of the costs of features are also performed. This information is fed back to the system architects and the management, who decide again if the new features will be implemented as system features or will remain in the scope of the specific deliverable system. In case some features are evaluated as system features, they are added to the system feature model and marked as pending. The next step is going towards designing and implementing the new features.

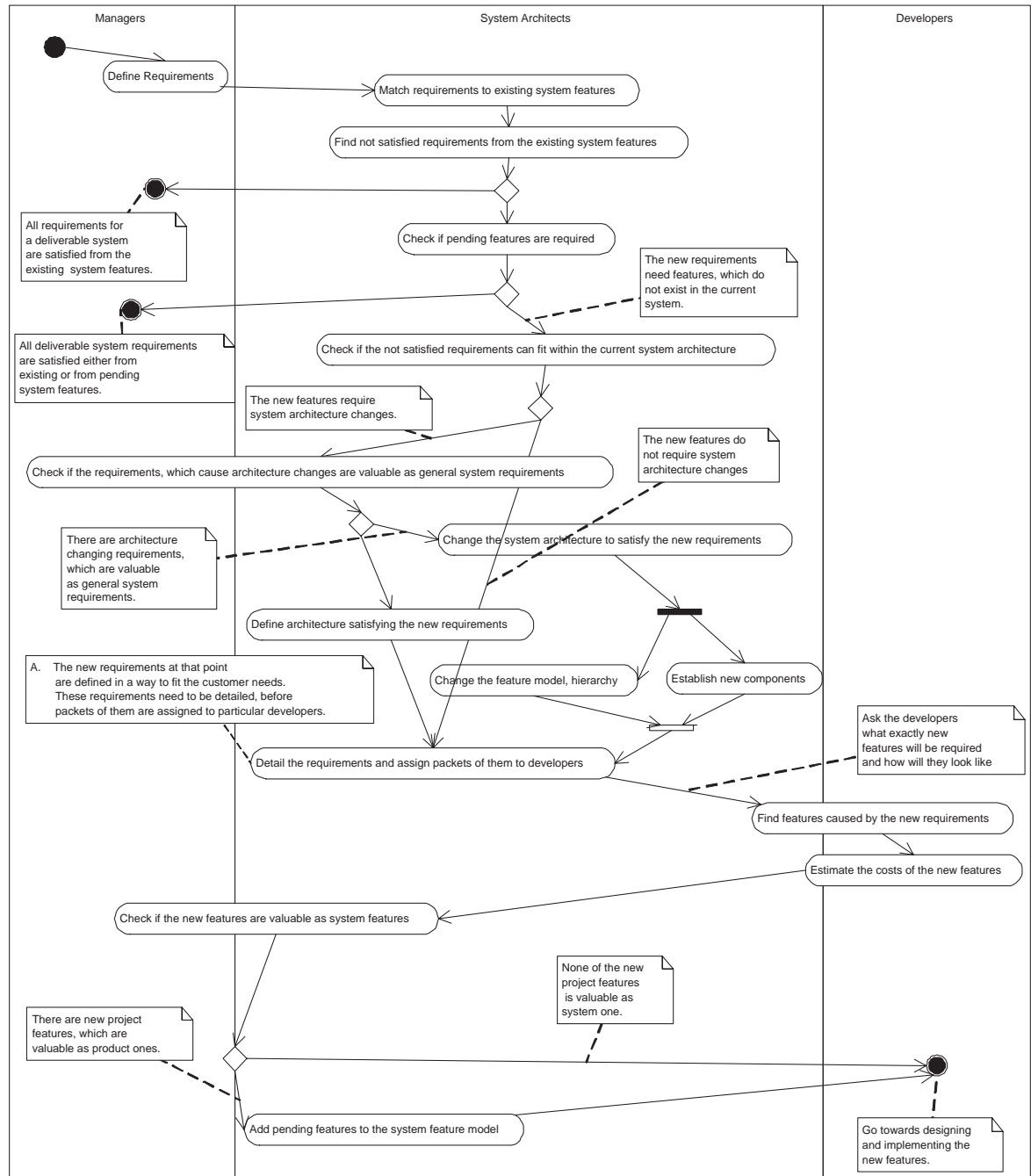


Figure 13.3: Feature-Based Requirements Engineering Activities

Several conclusions can be extracted from the feature-based requirements engineering activities and proved from the dataflow described in the next section:

- There is an active involvement of developers in the requirements engineering process.
- All stakeholders are involved only in activities requiring their expert knowledge.
- Overloading the stakeholders with unusual tasks is avoided.

13.4.3 Feature-Based Requirements Engineering Dataflow

On figures 13.4 and 13.5⁴ are presented top and detailed views of the requirements engineering dataflow. As shown in these figures, feature-based requirements engineering aims at using expert knowledge as much as possible. All stakeholders within the software engineering process contribute at their expert level to the software evolution and are able to influence the system architecture. At the same time, nobody is overloaded with inadequate work.

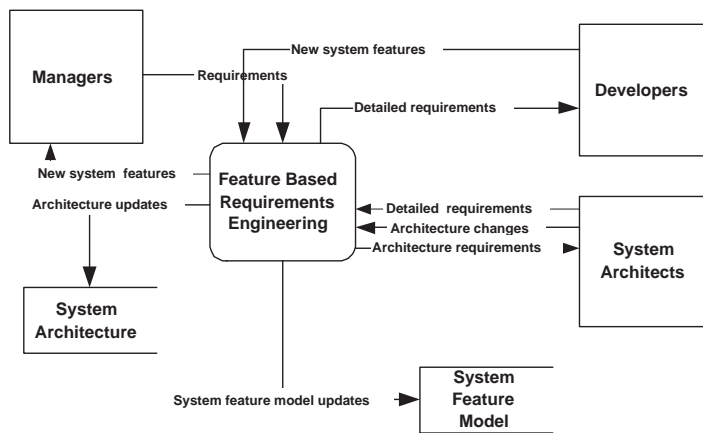


Figure 13.4: *Feature-Based Requirements Engineering Dataflow - Top View*

The dataflow of the feature-based requirements engineering process assures that only the necessary data is passed to the different stakeholders and overloading them with unnecessary details will be avoided. Managers deal only with information, which contains managing aspects, system architects take care of architecture and developers of implementation details. Nevertheless, they all have a common communication point, namely the feature model. Granting all of them, the opportunity to influence the feature model assures its consistency and compliance with latest system domain changes. On the other hand, the strict relation between features and architecture assures the update of the system architecture as soon as the feature model is changed.

For example, in figure 13.5, the management is concerned only with eliciting new system requirements and evaluating new system features. System architects process data of architecture changes and forward specific details concerning new system requirements to certain developers considered as experts in the subject problem. Developers process only the data specifically related to their responses and return their feedback. Due to this feedback, the developers have great influence on the feature model of the system. They are responsible for identifying the features implemented in the architecture components developed by them⁵. Still, the borders of the components are defined by system architects. They consolidate the information provided by the management and developers and keep the consistency of the system architecture.

⁴For the diagrams on figures 13.4 and 13.5, the Gane-Sarson DFD-Notations are used [Gane et al. 1979].

⁵Usually, the developers have the best expert knowledge concerning the features, which a software system has to provide in order to satisfy the variety of customer requirements. This statement is made according to the gained experience and the fact that the developers take part in the realization of all customer requirements for quite a lot of deliverable systems and are able to accumulate knowledge concerning the best way to satisfy new customer requirements. In that sense the developers are the most relevant stakeholders, which can identify a new feature supported by the system architecture. Nevertheless, it has to be taken into consideration that the developers are usually specialized in a specific area and can process correctly only requirements data concerning this area. For this reason the system architects dispatch the requirements data between the different developers.

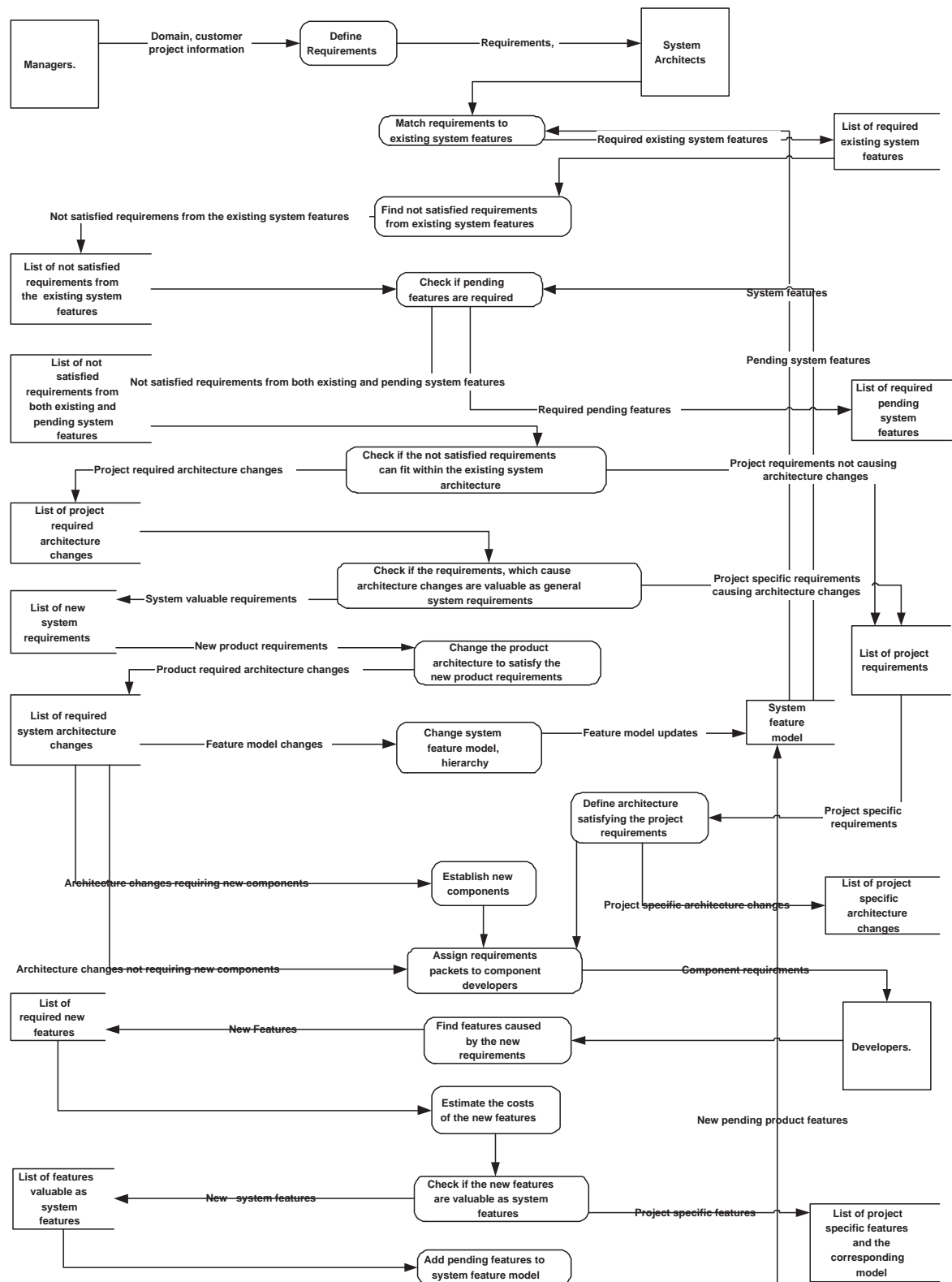


Figure 13.5: Feature-Based Requirements Engineering Dataflow - Detailed View

13.5 Software Evolution by Performing Feature-Based Requirements Engineering

The application of feature-based requirements engineering changes the software evolution process. As stated at the beginning of this chapter, the new requirements engineering approach aims at introducing early architecture refactorings, which, together with changes of the used architecture versioning strategy from complete architecture versioning to components versioning, will improve the software evolution process and prevent the decay of the system architecture. In this section, attention is paid to these topics.

13.5.1 The Changed Evolution Process

Figure 13.6 presents the software evolution process, which is caused by performing feature-based requirements engineering. As shown in the figure, the main principle of developing end-user deliverable system on the basis of system architecture is followed. The system architecture is reused in each new deliverable system. A new aspect caused by the use of feature-based requirements engineering is that the deliverable systems contribute to the evolution of system architecture and the dataflow between the system architecture and the deliverable systems is bidirectional. The deliverable systems are no longer just users of the system architecture as it was shown initially on figure 13.1⁶. In that sense, the difference between R&D⁷ projects and customer projects remains only in the volume of the made contribution. The way the two types of projects are handled is unified. All projects are based on the current system architecture and evolve it. The synchronization between the projects is performed through the feature model and the pending features mechanism. As a result, a conclusion can be made that the system evolves incrementally, increasing the number of implemented features and at the same time adapting the system architecture according to the relations between these features.

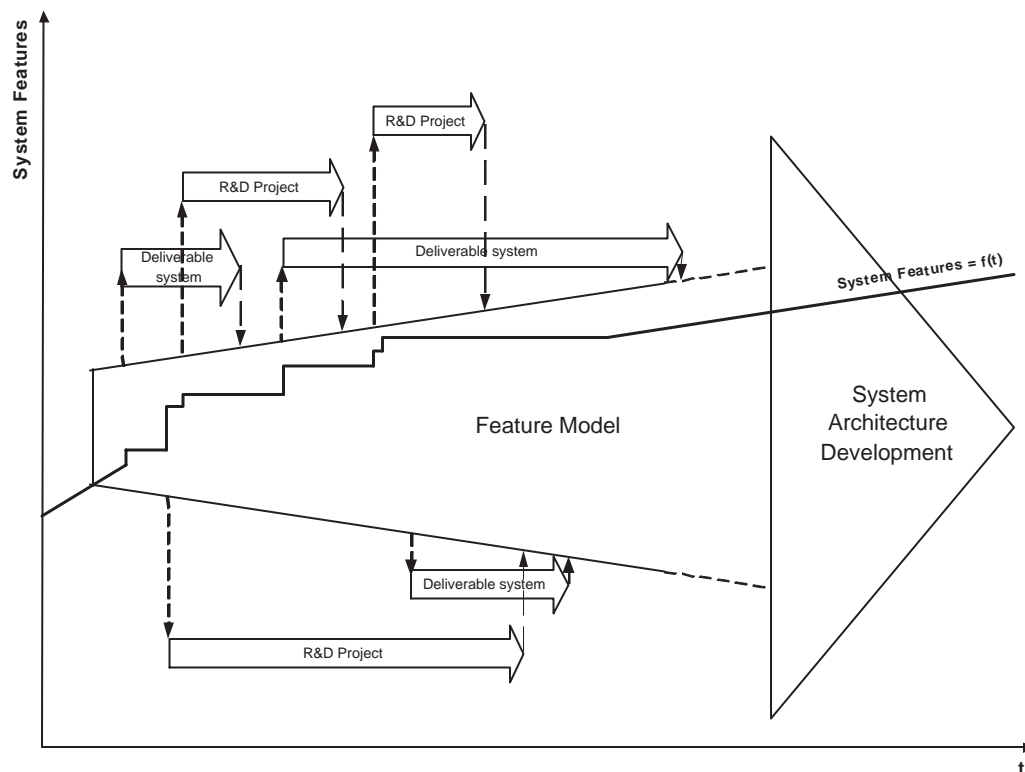


Figure 13.6: *Software Evolution Process by Performing Feature-Based Requirements Engineering*

⁶The dataflow shown on figure 13.1 is unidirectional, from the system architecture to the deliverable systems. No feedback from a deliverable system to the system architecture is considered.

⁷The R&D projects as initially defined have been the only once having bidirectional dataflow to the system architecture. These are the projects dealing especially with developing new features implemented from the system architecture.

13.5.2 Architecture Versioning Strategy

The evolution process caused by performing feature-based requirements engineering makes the architecture versioning strategy based on versioning the complete system architecture useless. Instead of versioning the system architecture as a whole, the approach introduces versioning of single architecture components (figure 13.7). Such a versioning strategy allows keeping the trace between features and component versions that implement these features.

The single component versioning strategy for system architecture is illustrated on figure 13.7. As shown, a component version is assigned to a set of features, i.e. a new architecture component or component version implements a set of new or changed system features. For example, the sets of features marked with "1" (features {F13, F14, F15}) and "2" (features {F3, F7, F8}) are implemented in the corresponding components "COMP A" and "COMP C 1.0". In a next version of the "COMP C" (version 1.1) a new feature is additionally implemented (F12). Thus the component "COMP C 1.1" implements a set of features marked with "3" on the figure (features {F12, F13, F14, F15}).

As pointed out at the beginning of this chapter, the component versioning strategy reduces the software maintenance costs. For example, to implement the three different sets of features shown on figure 13.7 by using complete architecture versioning strategy, it would have been necessary to maintain at least two different versions of the whole system architecture. In case the features have been implemented at a different time, the number of needed architecture versions would have been greater. The components versioning strategy requires maintaining only two different versions of just one component.

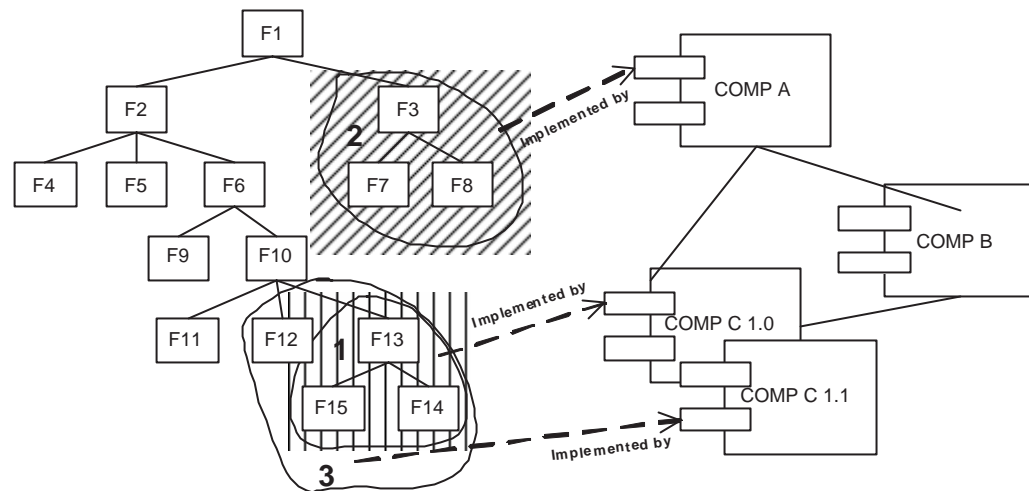


Figure 13.7: *Architecture Versioning Strategy by Using Feature-Based Requirements Engineering*

13.5.3 Integrated Architecture Refactoring (continuous migration)

One of the main tasks of the feature-based requirements engineering approach is to integrate architecture refactoring into the software evolution process, thus to realize the idea of applying early evolution changes. Such strategy should keep the system architecture up-to-date with the problem domain and as a result the architecture decay should be avoided or at least delayed. The following special features of the approach realize the refactoring ideas:

- Changes in the hierarchy of system features are allowed and are performed according to new system requirements.
- New features are also updated in the features hierarchy.
- Changes within the system architecture follow the changes of the features hierarchy.

The above-listed points assure that the feature model of the system will always present the actual system domain knowledge. If a feature, or respectively a system component, has become more or less important, this fact will be reflected in the feature model and the system architecture on time. As

a result, the system will be changed step by step and always kept up-to-date with the actual system domain requirements. One can say that there is a continuous small-step migration of the system to a new state (figure 13.8).

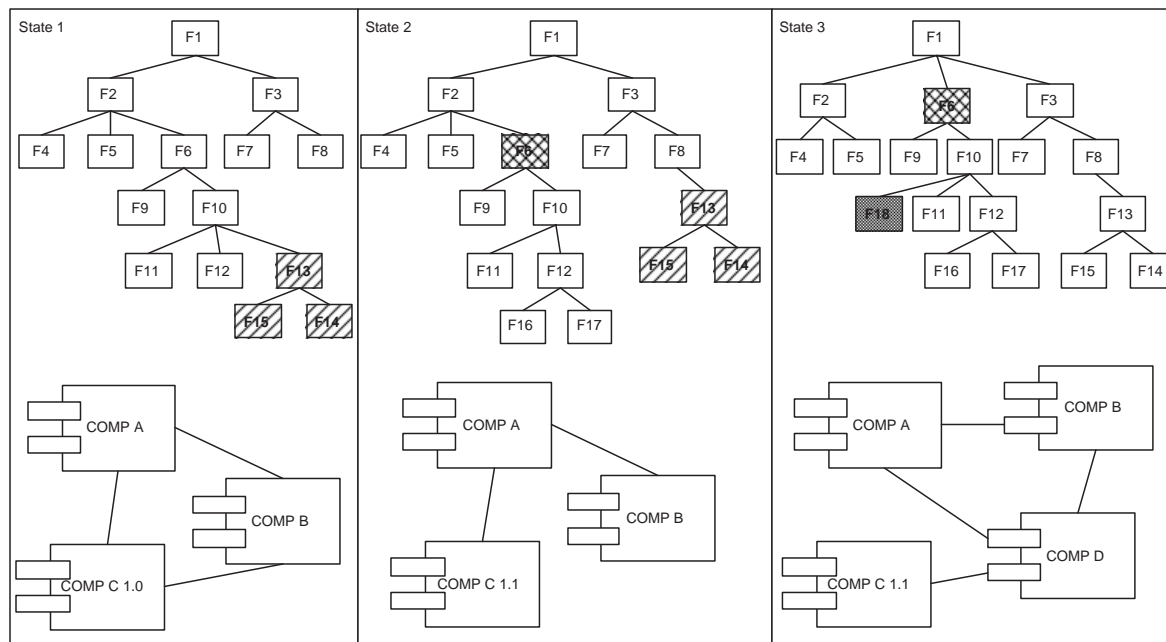


Figure 13.8: *Integrated Architecture Refactoring by Using Feature-Based Requirements Engineering*

For example, as it is shown in figure 13.8 the set of features from "State 1" {F13, F14, F15} has been moved on a higher hierarchical level at "State 2" of the feature model. Hence, this change has initiated changes to the system architecture as well. According to the example on figure 13.8 this change means a new version of "COMP C" and a removed dependence between "COMP B" and "COMP C". Similarly, the changes from "State 2" to "State 3" of the feature model (new position for the feature "F6" and new feature - "F18") have caused changes to the system architecture (new component - "COMP D" and changed dependencies between the components).

13.6 Conclusion

This chapter has presented a new approach for performing requirements engineering in software systems, which are a subject of evolution over a long period of time. A number of advantages make the feature-based requirements engineering an eligible methodology for supporting software evolution. These are:

- A continuous refactoring of the system architecture is introduced and thus the architecture decay is prevented.
- Persistence of the system knowledge is assured by the centralized feature model.
- The communication between stakeholders is significantly improved by using a common communication means - the feature model.
- Overloading the different stakeholders with unusual work is avoided. The approach assures that only the necessary information will be processed by the concerned experts.
- Due to the pending features synchronization mechanism, redundancies and double work are avoided.
- The approach is not a high-risk one. It does not require additional work, except for maintaining a system feature model and the traceability links between features and architecture components. The advantages of the approach are reached by more proper separation of the requirements engineering activities among the concerned stakeholders.

Similar to the feature-assisted reverse engineering and the feature-assisted architecture redesign, the feature-based requirements engineering lacks professional tool support. This can be considered a disadvantage of the approach, which can hinder its application. Nevertheless, there are certain possibilities to avoid the tools support problem with utilizing existent requirements engineering tools. For example, it is possible to use the Rational Requisite Pro [Rational 2003]. The tool is developed for supporting the requirements engineering activities defined from the Rational Unified Process[Rational 2003], but customization of the tool for the needs of the feature-based requirements engineering is also possible. Using different types of requisite pro requirements, feature and architecture model of a system can be established and the traceability links between them can be maintained. A part of the case study presented in chapter 14 is performed using the support of the Rational Requisite Pro tool.

The feature-based requirements engineering approach is the last from the package of methods developed in this thesis for supporting evolution of long-life software systems. The approach is the final node of the solutions aiming at preventing the decay of software architectures, increasing the lifetime of software systems and reducing maintenance costs.

Feature-based requirements engineering elaborates the idea of more intensive application of feature modelling in the software engineering process. It uses the foundations laid by feature-assisted reverse engineering and feature-assisted architecture redesign and utilizes the results of these methods for forward engineering tasks.

Part IV

Proof of Concept

Chapter 14

Applying Feature-Based Techniques within a Postal Automation System - Case Study

14.1 Motivation

Many of the ideas presented in this thesis have been developed and proved within a project in cooperation with the Reading Coding department of the Postal Automation division of Siemens Dematic AG in Constance, Germany. This project was used as a test field for the developed methodology for supporting software evolution. On the other hand, the project work was inspired by real needs of the industrial partner, which have become a guide for the developed ideas.

The case study presented in this chapter describes the results of applying the scientific research performed in this thesis in a real industrial environment. The aim of the study is to prove or reject the concepts stated by the new feature-based methods presented in chapters 11, 12 and 13. The application of the methods in a real industrial environment reveals the strengths and the shortcomings of the new approach. It clarifies the directions for future research as well.

14.2 Restrictions

The data used in the case study contains company confidential information for Siemens Dematic AG. For this reason it is not possible a full description of the case study to be published in the public version of this thesis. Such one is present only as an appendix, which will remain confidential and will be presented only in the evaluation copies of the work. This chapter¹ contains a short description of the case study illustrating the successful application in an industrial environment of the feature-based methodology for supporting software evolution.

14.3 Introduction

14.3.1 The Industrial Environment

Siemens Dematic AG is a company with more than 50 years experience in the domain of postal service logistics. A significant part of the products developed from the company is the Reading Coding system or shortly called the RC system.

The RC system provides services for automated recognition and manual coding of addresses and other mail related data. A typical architecture of such system is shown on figure 14.1. As visible on the figure, usually a reading-coding system is composed of three main components: flow management system, automated address reader and a pool of manual address coding desks. The flow management system is the heart of a RC system. It administers the flow of mail images, automated recognition

¹The public version of the case study

results and manual coding results. Additionally, the flow management system provides sorting commands to a sorting machine, as long as sufficient sorting information is collected. The automated reading is an OCR (optical character recognition) technology-based system. It attempts to recognize automatically the whole information located on a mail image. Images, which are failed to be processed from the automated reading are forwarded to a manual coding desk. There, the mail-related information is manually coded from an operator.

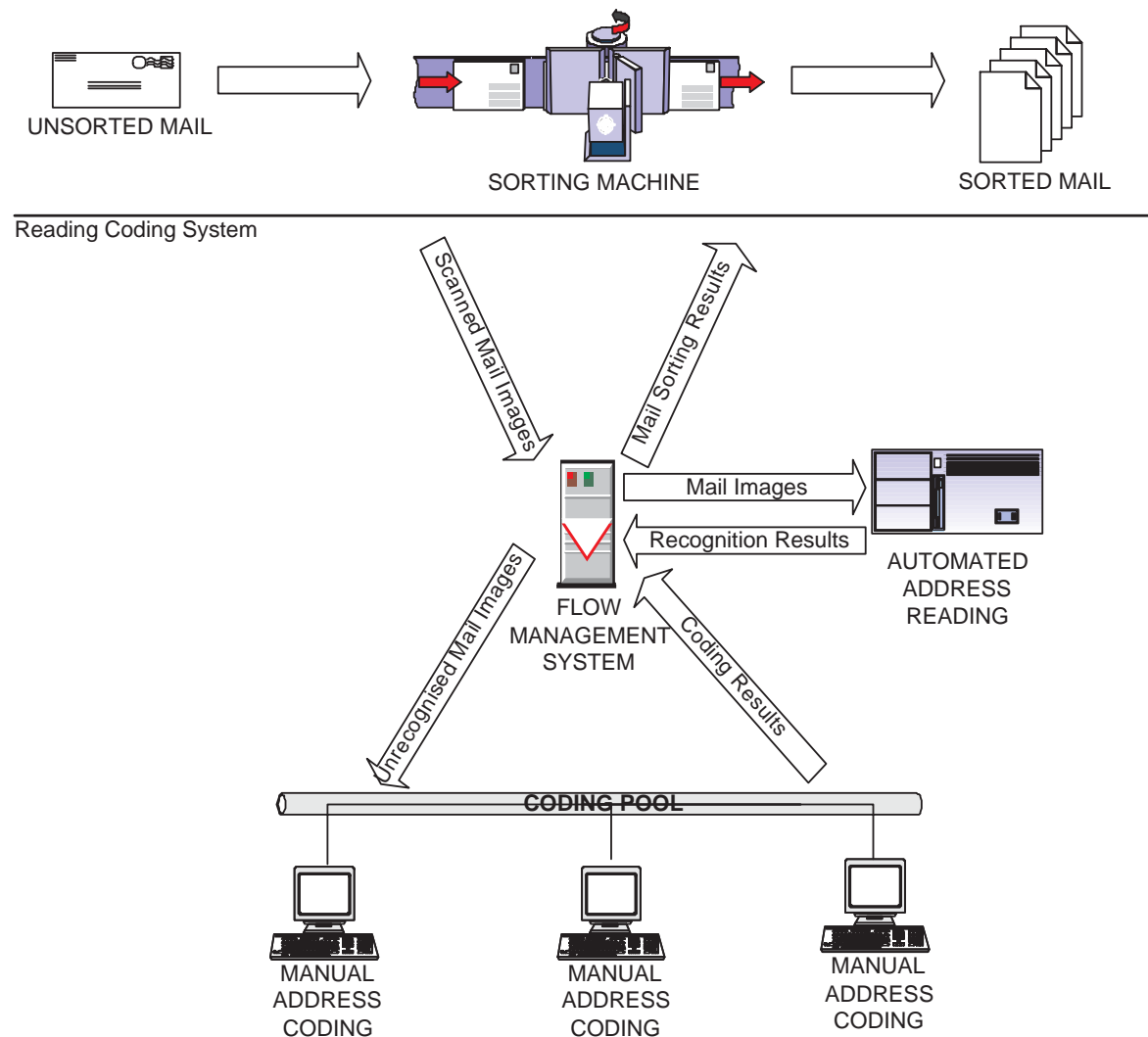


Figure 14.1: Typical Architecture of a Reading-Coding System

The history of the Siemens Dematic RC system dates back from the early 1990s. During its life time the RC system has been subject to many changes. Nevertheless, some main goals have been always considered for the system design, namely:

- Portability,
- Standardization,
- Modularity and Extensibility,
- Reusability and Flexibility,
- Simplicity,
- Scalability.

These design goals as well as some hard requirements for real time performance and reliability of the system have determined its development environment. Windows NT and Solaris have been chosen as operating system platforms. C, C++ have been chosen as portable development platforms, which can also satisfy the needs of the RC system for real-time processing. TclTk, Python and Visual Basic have been chosen as development platforms allowing quick prototyping and simplicity of the system configuration and the user interface. The complexity of the system has resulted in about 4 000 000 LOC source code.

As a market leader within the postal automation domain, Siemens Dematic has to deliver customized versions of the RC system satisfying a variety of country specific requirements. This fact determines the need of performing frequent significant changes to the RC system. Consequently, over the years it has become very difficult for the system to meet some of its initial design goals like modularity, extensibility, reusability, and flexibility. For this reason reengineering of the RC system architecture has been recently required. On the other hand, the people at the company responsible for this activity have realized, that the architecture reengineering is going to be much more effective for the company if additionally changes to the software engineering process of the RC system are introduced. These changes should help the company to maintain the RC system architecture up to date for a longer period of time.

The above circumstances have determined the perfect time and environment for introducing and validating the feature-based methodology developed in this thesis.

14.3.2 Challenges to the Feature-Based Methodology

The feature-based methodology had to meet a number of challenges within Siemens Dematic AG.

First, it had to support the architecture reengineering of the RC system. Although the system is well documented, as it often happens in the practice the system documentation has become outdated in many points. For this reason, it has been difficult to get an overview of the complete RC system architecture. An important requirement for the feature-based methodology was to provide a comprehensive overview of the RC system architecture in a short time.

Another challenge for the feature-based methodology concerning the RC system architecture reengineering was to support the work of the system architects. The methodology had the task to show them important redesign points likely to be missed. It had to give them a proof for some of their concerns about weaknesses of the architecture and had to provide them with a proposal how to define the necessary architecture components.

Finally, the feature-based methodology had to provide a way, which can help the system architects to prevent future decay of the RC system architecture, caused by the rapidly changing requirements to it.

14.4 Feature-Assisted Architecture Reengineering of an Industrial Software System

The feature-assisted reverse engineering and feature-assisted architecture redesign methods were used for supporting the reengineering of the RC system architecture.

Initially, as described in section 11.2, a feature model describing the architecture was obtained following the feature-assisted approaches. The model contains both design objectives and design decisions for the components composing the architecture. The model provided the necessary overview of the properties and the tasks of the architecture. External reviewers used the model as an input supplying them with the necessary information for performing an architecture review.

In a second step, the descriptive model was used as a basis for performing disproportions and redundancies analyses (see sections 12.3 and 12.5). As proposed from the feature-assisted architecture redesign method (see figure 12.1), the features composing the model were clustered (see section 12.4). The clustering results together with the ones from the previous two analyses were used from the system architects as clues by their architecture reengineering work. All analyses results were presented and positively evaluated in an architecture workshop held together with experts from the Siemens Dematic RC system team.

Both the feature-assisted reverse engineering and the architecture redesign methods were applied for the RC system as it was described in chapters 11 and 12. No changes to the methods were required

in order to apply them in the industrial environment. The concrete results of the application of the methods for the RC system are presented in the appendix part of the thesis.

14.5 Performing Feature-Based Requirements Engineering in an Industrial Environment

As it was mentioned in the introduction of this chapter, the feature-based methodology developed in this thesis had to help the RC system architects to prevent future decay of the RC system architecture. The feature-based requirements engineering was recognized by them as an eligible method for this task.

In order to apply the feature-based requirements engineering for the RC system, the process defined in chapter 13 (see figures 13.3 and 13.5) had to be slightly changed. The most significant changes were related to the structure of the used feature model² in order to make its interface more suitable for the management of the company. In addition, considering the specifics of the RC system engineering process, the feature-based requirements engineering process was slightly modified allowing reuse of requirements that have been defined for different projects. Also, some changes of the stakeholders' roles were necessary. Due to legal restrictions, these changes are described in the applied appendix.

Despite the performed changes, the basic ideas of the feature-based requirements engineering were applied for the RC system engineering process as described in chapter 13 (see figure 13.6). The approach was introduced to the different concerned stakeholders in a number of presentations and coaching sessions. The new process was generally well accepted from both the system architects and the management.

The good acceptance of the feature-based requirements engineering method can be considered especially valuable. It proves the advantages of the approach. Nevertheless, the real value of the approach can be evaluated only over a longer period of time. The complete success of the approach will be reached in case no new architecture reengineering, i.e. no "revolution" is required for the RC system during this decade.

14.6 Tool Support

There are certain tools, which were applied in this case study, although the tool support for the feature-based methodology is generally insufficient.

The feature model of the RC system was initially created using the AmiEddi tool [AmiEddi]. AmiEddi provides a convenient graphical environment for feature modeling. Unfortunately, integration of the tool with other tools is difficult and ineffective. Also, the reporting capabilities of AmiEddi are quite insufficient.

As long as the feature model of the RC system was established and verified, the data was transferred from AmiEddi into a MS Access database. Over this database the tool support for the feature-assisted architecture redesign was developed (see chapter 12). This tool support was used for performing the architecture disproportions analysis, redundancies analysis and for feature clustering. As described in chapter 12, the information retrieval required in the feature-assisted architecture redesign was performed using the PolyAnalyst[PA2003] tool.

Since the tool supporting the feature-assisted architecture redesign is no requirements engineering tool, another transformation of the feature model data was necessary. In order to apply the feature-based requirements engineering in the RC system environment, the feature model of the system had to be transformed into a Rational Requisite Pro [Rational 2003] requirements model. Then, as described in chapter 13 with the help of the Rational Requisite Pro tool the realization of the feature-based requirements engineering ideas was possible.

14.7 Conclusion

The case study illustrating the successful application of the feature-based methodology in a real industrial environment can be considered as a proof of concept for the methodology. The positive evaluation of the methods and their results, both from the development and the management of the studied RC

²Normally, the feature-based requirements engineering utilizes feature models having the structure described in section 10.6.

system proved the value of the work developed in this thesis. Also, the cooperation with the industrial partners revealed the strengths and the shortcomings of the approach.

The main critique to the feature-assisted reverse engineering and the feature-assisted architecture redesign methods remains the insufficient tool support for them. Despite this, the feature-based approaches have been well accepted by the RC system team, which proves the value of the methodology behind them. The established RC system feature model found good acceptance among development and management. The results of the feature-assisted architecture redesign method were evaluated by the RC system architects as a good help for their work. The feature-based requirements engineering process is about to find its full acceptance among all concerned parties and to become a part of the RC system engineering process.

Part V

Conclusion

Chapter 15

Summary

The goal of the work presented in this thesis was to develop a methodology for supporting the evolution of long-life software systems. The work is performed on the basis of a number of areas from the software engineering, which have been discussed in part 2 of the thesis. In that sense, a significant contribution of the thesis is that it summarizes some of the most important software engineering technologies concerning the software evolution. These technologies are:

- The software life cycle models, which determine the framework for software evolution;
- The domain engineering, which collects and implements knowledge for reuse;
- The feature modeling, which is a technology allowing accumulation and well structured presentation of important system concerns and thus has become the key technology utilized in the thesis;
- Re-engineering, which provides techniques for analyzing, understanding and reworking software systems, activities which are an inevitable part of the software evolution process;
- Software architecture, which is the design base for overcoming the complexity of software systems and which determines the ability of the software systems to evolve;
- Software product lines and the related requirements engineering technology, which are currently the state-of-the-art solution for evolution of long-life software systems.

Considering the important aspects of the previously mentioned technologies, in part 3 of this thesis a methodology for supporting the software evolution has been developed. This methodology comprises three methods. Two of them, namely the feature-assisted reverse engineering and the feature-based architecture redesign present a technical solution supporting the work of the software engineers. These methods help system architects to perform important evolution tasks concerning system reengineering.

The feature-assisted reverse engineering, presented in chapter 11 is an approach for supporting program understanding and recovery of both static and dynamic aspects of software architecture. It utilizes the feature modeling technology for those tasks. The approach develops a comprehensive procedure for collecting and modeling system features and use cases and for additional verification of the established models. On the basis of the provided solid base, feature-based program understanding and feature-based architecture recovery are performed.

The feature-based architecture redesign, presented in chapter 12 is an approach for supporting the work of the software engineers by redesigning software architectures. The approach provides the software engineers with good clues and hints about architecture redesign activities. This information is obtained by analyzing the problem domain knowledge gathered in a feature model of a software system. The approach also demonstrates how analyses performed over feature models and application of information retrieval techniques can hint the software engineers existing design problems such as unbalanced architecture and design redundancies. In addition, the proposed analyses provide suggestions for overcoming those problems. A step ahead towards assuring completeness and appropriateness of architecture redesign is the provided feature-based mechanism for mapping between old and new architecture solutions.

The feature-assisted reverse engineering and the feature-based architecture redesign methods increase the effectiveness of the changes concerning the software architecture. Thus one of the initial

aims of this thesis is reached. Still, changes are an inevitable part of the software evolution. As it was stated in part 1 of this thesis, it is better to avoid performing high risk changes in a software system in order to increase its evolvability. An appropriate way to reach such a goal is to provide a solution for supporting the software evolution, which concerns the organizational aspects of the software engineering. Such a solution is developed by the feature-based requirements engineering, presented in chapter 13.

The feature-based requirements engineering presents an approach for forcing continuous refactoring of the software systems' architecture. In comparison to architecture reengineering, which is often used in the practice, refactoring is a low risk technique. By using refactoring, timely updates of the system architecture are assured. Thus, the architecture is kept up-to-date with the requirements of the problem domain. Consequently, the need of performing high risk changes is avoided for a long period of time, which was one of the initial aims of this thesis. The feature-based requirements engineering approach also significantly improves the communication between the stakeholders in a software system and assures dataflow, which avoids overloading them with work out of their duties. All these properties of the approach make it an important contribution to the methodological support of the software evolution.

The complete methodology for supporting software evolution including the feature-assisted reverse engineering, the feature-based architecture redesign and the feature-based requirements engineering has been developed in cooperation with an industrial partner, namely the postal automation division of Siemens Dematic in Constance, Germany. A case study (presented in part 4) performed over the system developed from the industrial partner has proven the value of the methodology and has helped it to find its first acceptance in the real software development world.

Chapter 16

Future Work

The work presented in this thesis has outlined the open questions, which stay before the researchers aiming at inventing a methodological solution for software evolution. A number of issues have to be considered, namely:

- The knowledge from multiple areas of software engineering has to be consolidated and a technological base for developing highly evolvable software systems has to be established. Although this thesis has made a significant contribution in consolidating the knowledge concerning software evolution, it can not claim to cover all related areas. In order to complete such a task, cooperation between researchers and software engineers from all different areas of the software engineering field is required.
- This thesis has revealed the power of applying information retrieval techniques for software evolution tasks. Still, a large area of undiscovered solutions concerning this topic is opened and worth getting attention. For example, the application of information retrieval techniques over system documentation could reveal information showing hidden dependencies within a software system. Similarly, the information retrieval techniques can support recovery of the design decisions history in a software system. This information is especially valuable for the software evolution.
- Further researching the role of the feature modeling in the software engineering process is worth being considered. This thesis has made a step ahead showing the power of this technology by using it for software evolution tasks. It was also shown, how with the help of the feature modeling, the unnatural link between requirements and design elements can be broken with features. Still, there is a wide field of open issues concerning the role of feature modeling in the software engineering process. The influence of system features over the source code and the system implementation needs to be further investigated.
- A technology can find its proper application in the real world only if sufficient tool support for it is assured. As mentioned, the tools developed and used within the frame of this thesis are rather prototype ones. They outline some of the basic requirements for tool support for the feature-based methodology, namely:
 - The feature modeling tools have to distinguish between design objectives and design decisions features;
 - Feature modeling tools need to provide the possibility for exporting the feature models in a standard database format, which is accessible for information retrieval tools.
 - Traces between requirements, features and architecture components have to be maintained;
 - Feature modeling tools have to be easy integrable with requirements engineering and architecture modeling tools;
 - Feature modeling tools have to support definition of feature attributes in order to distinguish between different kinds of features. For instance, already implemented features and features whose implementation is still pending should be distinguished in a feature model.

Nevertheless, the above listed items present only the basic requirements to a comprehensive tool support for the feature-based methodology developed in this thesis. The definition of the precise requirements is a challenge, which has to be met in order to assure the basis for the industry to develop the needed tools.

- The work in this thesis has shown that software evolution is a constructive process, which can be much more effective in case the "revolutions" in it are avoided. The only way to reach such a result is by changing the way of thinking of the people responsible for the software systems engineering. The software engineering theory has to find the way to provoke strategic thinking and to minimize the chaotic daily work, which is nowadays often met in many software developing institutions. In order to meet this challenge the software research has to pay more attention to the organizational aspects of software engineering instead of providing an avalanche of technical solutions, which find harder and harder acceptance in the real world.

Bibliography

- [AM] Manifesto for Agile Software Development, <http://www.agilemanifesto.org/>
- [AmiEddi] Generative Programming WWW site, <http://143.93.17.153/gp-org/download/index.php>
- [Arrango 1994] Arrango G.: Domain Analysis Methods. In Software Reusability, Schfer, R. Prieto-Daz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, 1994, pp. 17-49
- [Atkinson et al. 2000] Atkinson C., Boger J., Mithig D.: Component-Based Product Line Development: The Kobra Approach. In Software Product Lines Experience and Research Directions, pages 289-309, Kluwer Academic Publishers, 2000
- [Balazinska et al.] Balazinska, M.; Merlo, E.; Dagenais, M.; Lage, B.; Kontogiannis, K.: Advanced clone-analysis to support object-oriented system refactoring. Proceedings 7th Working Congress on Reverse Engineering (WCRE 2000). Brisbane 2000. Computer Society Press, 2000, pp. 98-107.
- [Baker 2002] Baker S.: On Finding Duplication and Near-Duplication in Large Software Systems. In: Wills, L., Newcomb P. and Chikofsky E.: Second Working Conference on Reverse Engineering, Sites 8695, Los Alamitos, California, Juli 1995. IEEE Computer Society Press.
- [Bass et al. 1998] Bass, L., Clements, P., Kazman, R., Software Architecture in Practice, Addison-Wesley, 1998.
- [Baxter et al. 1998] Baxter I., Yahin A., Moura L., Santanna M., Bier L.: Clone Detection Using Abstract Syntax Trees. In: Proceedings; International Conference on Software Maintenance, 1998.
- [Beck 1999] Beck K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, 1999
- [Bellon 2002] Bellon S.: Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Master Thesis, Institute for Informatics, University of Stuttgart, 2002 (in german).
- [Bengtsson et al. 1998] Bengtsson, P., Bosch, J.: Scenario-Based Software Architecture Reengineering, Proc. 5th International Conference on Software Reuse (ICSR5), pp.308-317, IEEE Computer Society Press, Victoria, B.C, Canada, June 1998.
- [Bennett et al. 2000] Bennett K., Rajlich V.: Software Maintenance and Evolution: a Roadmap. "The Future of Software Engineering" , Anthony Finkelstein (Ed.), ACM Press 2000
- [BH03] Project Bauhaus Software Architecture, Software Reengineering, and Program Understanding. <http://www.bauhaus-stuttgart.de/bauhaus/index-english.html>
- [Boellert 2002] Boellert, K., Objektorientierte Entwicklung von Software-Produktlinien zur Serienfertigung von Software-Systemen. Phd-Thesis Technical University of Ilmenau (in german).
- [Boehm 1988] Boehm B.W.: A Spiral Model of Software Development and Enhancement. IEEE Computer 21, 1988
- [Boger et al. 1999] Boger et al.: PuLSE: A Methodology to Develop Software Product Lines. In Proceedings of the 5th Symposium on Software Reusability (SSR99), pages 122-131. ACM Press, 1999

- [Boger et al. 2002] Boger, M.; Sturm, T.; Fragemann, P.: Refactoring browser for UML. In Proc. 3rd Intl Conf. on eXtreme Programming and Flexible Processes in Software Engineering, 2202, pp. 77-81, Alghero Italy.
- [Booch 1994] G. Booch, Object-Oriented Analysis and Design with Applications, Redwood City, California, The Benjamin/Cummings Publishing Company, Inc., 1994.
- [Bosch 2000] Bosch, J., Design & Use of Software Architectures, Addison-Wesley, 2000.
- [Brooks et al. 1978] Brooks, R.: Using a Behavioral Theory of Program Comprehension in Software Engineering, Proc. 3rd Int. Conf. on Software Eng. New York: IEEE, 1978
- [Brooks et al. 1982] Brooks, R.: A Theoretical Analysis of the Role of Document. in the Comprehension of Computer Programs. Proc. Conf. on Human Factors in Computer Systems. New York: ACM, 1982
- [Brooks et al. 1983] Brooks, R.: Towards a Theory of the Comprehension of Computer Programs. Intl. J. Man-Machine Studies 18, 6 (June 1983)
- [Buschman et al. 1996] Buschman F., Meunier R., Rohnert H., Sommerlad P., Stal M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley Sons; 1 edition (August 8, 1996)
- [Cameron et al. 1994] Cameron, R., Ito, M.: Grammar-Based Definition of Meta-Programming Systems. ACM Transaction of Programming Languages and Systems Vol. 6, No. 1, January 1994, Pages 20-54
- [Canfora et al. 1994] Canfora, G., De Lucia, A., di Lucca, G. Fasolino, A.: Recovering the Architectural Design for Software Comprehension, Proc. IEEE Third Workshop on Program Comprehension, Washington, DC, November 1994.
- [CARDS 1994] Software Technology For Adaptable, Reliable Systems (STARS). Domain Engineering Methods and Tools Handbook: Volume I Methods: Comprehensive Approach to Reusable Defense Software (CARDS). STARS Informal Technical Report, STARS-VC-K017R1/001/00, December 31, 1994, <http://nsdir.cards.com/libraries/HTML/CARDS-documents.html>
- [Carriere et al. 1999] Carriere, S. J., Kazman, R., Woods, S.G., "Assessing and maintaining. architectural quality", Proceedings of the 3rd European Conference on Software Maintenance and Reengineering, pp. 22-30, 1999.
- [Clayton et al. 1997] Clayton, R., Rugaber, S., Taylor, L., Wills, L.: A Case Study of Domain-based Program Understanding. 5th Workshop on Program Comprehension, Dearborn, Michigan, 1997
- [Clayton et al. 1998] Clayton, R., Rugaber, S., Wills, L.: On the Knowledge Required to Understand a Program. The Fifth IEEE Working Conference on Reverse Engineering'98, Honolulu, Hawaii, October 1998
- [Clements 1996] Clements P.: A Survey of Architecture Description Languages. 8th International Workshop in Software Specification and Design, Germany, 1996
- [Corbi et al. 1989] Corbi, T. A.: Program Understanding: Challenge for the 1990's. IBM Systems J. 28, 1989
- [Czarnecki et al. 2000] Czarnecki K., Eisenecker U.: Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models, Addison Wesley, 2000
- [DSDM] Dynamic Systems Development Method, <http://www.dsdm.org>
- [Ducasse et al. 1999] Ducasse S., Rieger M., Demeyer S.: A Language Independent Approach for Detecting Duplicated Code. In: Proceedings of the International Conference on Software Maintenance (ICSM99), 1999.

- [Eisenbarth et al. 2001] Eisenbarth T., Koschke R., Simon D.: Aiding Program Comprehension by Static and Dynamic Feature Analysis. In: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, November 2001.
- [Eisenbarth et al. 2003] Eisenbarth T., Koschke R., Simon D.: Locating Features in Source Code, IEEE Transactions on Software Engineering, Vol. 29, No. 3, March, pages 195-209.
- [Erni et al. 1996] Erni, K.; Lewerentz, C.: Applying Design-Metrics to Object-Oriented Frameworks. In Proceedings of the 3rd International Software Metrics Symposium (March 25 - 26, 1996, Berlin, Germany), IEEE Computer Society Press, 1996, pp. 64 - 74.
- [Faure et al.] Faure, D.; Ndellec, C.: A corpus-based Conceptual Clustering Method for Verb Frames and Ontology. In LREC workshop on Adapting lexical and corpus resources to sublanguages and applications, Granada, Spain, Mai 1998.
- [FDD] Feature Driven Development, <http://www.nebulon.com/fdd/index.html>
- [Fere et al. 1999] Fere X., Vegas S.: An Evaluation of Domain Analysis Methods. 4th CAiSE/IFIP8.1 International Workshop in Evaluation of Modeling Methods in Systems Analysis and Design(EMMSAD99), Heidelberg, Germany
- [Finnigan et al. 1997] Finnigan, P., Holt, R., Kalas, I., Kerr, S., Kontogiannis, K., Mller, H., Mylopoulos, J., Perelgut, S., Stanley, M., Wong, K.,.: The Software Bookshelf, IBM Systems Journal, Vol. 36, No. 4, pp. 564-593, November 1997.
- [Fowler 1999] Fowler M.: Refactoring: Improving the Design of Existing Code, Addison-Wesley Object Technology Series, 1999.
- [Gamma et al. 1995] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object Oriented Design. Addison Wesley, 1995
- [Gane et al. 1979] Gane, C., Sarson. T.: Structured Systems Analysis. Englewood Cliffs, NJ.: Prentice-Hall, 1979.
- [Garlan et al. 1993] Garlan D., Schon M.: An Introduction to Software Architecture. In proceedings of "Advances in Software Engineering and Knowledge Engineering", Singapur, 1993
- [Garlan 2000] Garlan D.: Software Architecture: A Roadmap. The Future of Software Engineering, ACM Press, pp. 91-101, 2000
- [Gilb 1988] Gilb T.: Principles of Software Engineering Management. Addison Wesley Publishing Co., 1988
- [Griss et al. 1998] Griss M., Favaro J., dAlessandro M.: Integrating Feature Modeling with the RSEB. In Proceedings of the Fifth International Conference on Software Reuse (Victoria, Canada, June 1998), pp. 76-85, IEEE Computer Society Press, 1998, see <http://www.intecs.it>
- [Hautus 1999] Hautus, E.: Improving Java Software through Package Structure Analysis. In F. Balmas, M. Blaha, and S. Rugaber, editors, Proceedings WCRE'99 (6th Working Conference on Reverse Engineering). IEEE, Oct. 1999.
- [Highsmith et al. 2000] Highsmith III J., Orr K.: A Collaborative Approach to Managing Complex Systems. Dorset House, 2000
- [Highsmith 2002] Highsmith J.: Agile Software Development Ecosystems. Addison Wesley, 2002
- [Higo et al. 2002] Higo Y., Ueda Y., Kamira K., Kusumoto S., Inoue K.: On Software Maintenance Process Improvement Based on Code Clone Analysis. Proc. 4th International Conference, PROFES 2002 Rovaniemi, Finland, December 2002. LNCS 2559, Springer, 2002, pp.185-197
- [Hofmann 2000] Hofmann, H., Requirements Engineering, Deutscher Universitts-Verlag.
- [Hofmeister et al. 2000] Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison Wesley, 2000.

- [Hyatt et al. 1996] Hyatt, L., Rosenberg, L.: "A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality," 8th Annual Software Technology Conference, UT, 4/96.
- [Ibanez et al. 1996] Ibanez, M. ; Rempp, H., European User Survey Analysis, ESPITI project report.
- [IEEE Std 1471-2000] IEEE Recommended Practice for Architecture Description of Software Intensive Systems, IEEE Computer Society, 2000
- [Jacobson et al. 1992] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, Object-Oriented Software Engineering, Workingham, England, Addison-Wesley, 1992.
- [Jacobson 1994] Jacobson I., The Object Advantage - Business Process Reengineering with Object Technology, Addison-Wesley, Menlo Park, CA, 1994.
- [Jacobson et al. 1997] Jacobson I., Griss M., Jonsson P., Software Reuse: Architecture, Process and Organization for Business Success, Addison-Wesley-Longman, May 1997.
- [Johansson et al. 2002] Johansson E., Martin Hst M., Tracking Degradation in Software Product Lines through Measurement of Design Rule Violations, 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), Ischia, Italy, July, 2002.
- [Kamya et al.] Kamiya T., Kusumoto S., Inoue K.: CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. IEEE Trans. Software Engineering (to appear).
- [Kang et al. 1990] Kang, Kyo C.; Sholom G. Cohen, J. A. Hess, W.E. Novak, A. Peterson, S., Feature-Oriented Domain Analysis (FODA): Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
- [Kazman et al. 1997] Kazman, R., Carriere, S.: Playing Detective: Reconstructing Software Architecture from Available Evidence. Software Engineering Institute, Carnegie Mellon University, CMU/SEI-97-TR-010/ESC-TR-97-010
- [Kazman et al. 1998a] Kazman, R., Carriere, S. J.: View Extraction and View Fusion in Architectural Understanding, Proc. 5th International Conference on Software Reuse (ICSR5), pp.290-299, IEEE Computer Society Press, Victoria, B.C, Canada, June 1998.
- [Kazman et al. 1998b] Kazman, R., Klein, M., Barbacci, M., Lipson, H., Longstaff, T., Carriere, S. J.: The Architecture Tradeoff Analysis Method, Proc. Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS98), pp.68-78, Monterey, USA, August 1998.
- [Klein et al.] Klein M., Kazman R.: Attribute-Based Architecture Styles, Technical Report CMU/SEI-99-TR-022, www.sei.cmu.edu
- [Krinke 2001] Krinke J.: Identifying Similar Code with Program Dependence Graphs. In: Proceedings of the Eighth Working Conference On Reverse Engineering (WCRE01), 2001.
- [Kontogiannis et al. 1995] Kontogiannis K., DeMori R., Bernstein M., Galler M., Merlo E.: Pattern matching for design concept localization. In: WCRE 95: Proceedings of the Second Working Conference on Reverse Engineering, (Toronto, Ontario; July 14-16, 1995), Seiten 96103. IEEE Computer Society Press (Order Number PR07111), Juli 1995.
- [Koskimes et al. 1995] Koskimes K., Mssenback H., Designing a Framework by Stepwise Generalization. 5th European Software Engineering Conference Barcelona, Lecture Notes in Computer Science 989, Springer, 1995.
- [Kotler et al. 1999] Kotler, Ph.; Bliemel, F., Marketing-Management: Analyse, Planung, Umsetzung und Steuerung. Schffer-Poeschel, 9th Edition (in German).
- [Kruchten 1995] Kruchten, P. B.: The 4+1 View Model of Architecture. IEEE Software, 12(6): 42-50, 1995

- [Kuusela et al. 2000] Kuusela, J., Savolainen, J.: Requirements Engineering for Product Families. ICSE 2000, Proc. 22nd Int. Conf. on Software Eng., Limerick Ireland. ACM, 2000
- [Letovsky et al. 1986] Letovsky, S., Soloway E.: Delocalized Plans and Program Comprehension. IEEE Software 3, 3 (May 1986)
- [Liao] Liao L.: From Requirements to Architecture: The State of The Art in Software Architecture Design
- [Linger et al.] Linger R., Lipson H., McHugh J., Mead N., Sledge C.: Life-Cycle Models for Survivable Systems, TECHNICAL REPORT CMU/SEI-2002-TR-026
- [Loucopoulos 1995] Loucopoulos P., System Requirements Engineering. McGraw Hill, 1995
- [Mancoridis et al. 1998] Mancoridis, S., Mitchell, B. S., Chen, Y., Gansner, E. R.: Bunch: A clustering tool for the recovery and maintenance of software system structures. Proceedings of ICSM, 1998.
- [Martin et al. 2001] Martin R., Schmalzer K., Beedle M.: Agile Software Development with Scrum(Agile Software Development). Prentice Hall, 2001
- [MBSE97] Software Engineering Institute. Model-Based Software Engineering. WWW pages, URL: <http://www.sei.cmu.edu/technology/mbse/>, 1997
- [Mens et al.] Mens, T.; Demeyer, S.; Du Bois, B.; Stenten, H. Van Gorp, P.: Refactoring: Current research and Future Trends. ETAPS workshop LDTA 2003
- [Monden et al. 2001] Monden, A.; Nakae, D.; Kamiya, T.; Sato, S.; Matsumoto, K.: Software Quality Analysis by Code Clones in Industrial Legacy Software. Information Science Technical Report, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-TR2001007, Sep. 2001.
- [Nelson 1996] Nelson M.: A survey of Reverse Engineering and Program Comprehension. ODU CS551-Software Engineering Survey, 1996
- [PA2003] PolyAnalyst, <http://www.megaputer.com/products/pa/index.php3>
- [Pashov et al. 2003] Pashov, I., Riebisch, M.: Using Feature Modeling for Program Comprehension and Software Architecture Recovery. In: Proceedings 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'03), Huntsville Alabama, USA, April 7-11, 2003. IEEE Computer Society, 2003.
- [Philippow et al. 2001] Philippow, I.; Riebisch, M., Systematic Definition of Reusable Architectures. Proceedings 8.th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, April 17-20, S. 128-136.
- [Philippow et al. 2003] Philippow I., Streitferdt D., Riebisch M.: Design Patterns Recovery in Architectures for Supporting Product Line Development and Application. In: M. Riebisch, J. O. Coplien, D. Streitferdt (Eds.): Modelling Variability for Object-Oriented Product Lines. BookOnDemand Publ. Co., Norderstedt, 2003. pp. 42-57.
- [Pree 1996] Pree W., Framework Patterns. White Paper, SIGS Books, New York.
- [Pressman 1991] Pressman R.: Software Engineering. MC Grow Hill Text, 1991
- [PTW] Reverse Engineering Tutorial.
<http://www.program-transformation.org/twiki/bin/view/Transform/ProgramTransformation>
- [Rajlich et al. 1994] Rajlich, V., J. Doran, Gudla R.: Layered Explanations of Software: A Methodology for Program Comprehension, Third Workshop on Program Comprehension, WPC'93, Washington, D.C., pp. 46-52, Nov. 1994
- [Rajlich et al. 2000] Rajlich V., Bennett K.: A Staged Model for the Software Life Cycle. IEEE Computer 33(7): 66-71 (2000)
- [Rational 2003] www.rational.com

- [REF] Refactoring. <http://www.refactoring.com/>
- [RETAX] Reengineering and Reverse Engineering Terminology, <http://www.tcse.org/revengr/taxonomy.html>
- [RISE 2003] Research Institute for Software Evolution, <http://www.dur.ac.uk/CSM/> (viewed 2003)
- [Riebisch et al. 2002] Riebisch, M.; Boellert, K.; Streitferdt, D.; Philippow, I.: Extending Feature Diagrams with UML Multiplicities. In Proc. of Integrated Design and Process Technology (IDPT) 2002, Session 4 pp.1-7.
- [Riebisch 2003] Riebisch M.: Towards a More Precise Definition of Feature Modeling. Position Paper. In: M. Riebisch, J. O. Coplien, D. Streitferdt (Eds.): Modelling Variability for Object-Oriented Product Lines. BookOnDemand Publ. Co., Norderstedt, 2003. pp. 64-76.
- [Riva et al. 2002] Riva, C., Rodriguez, J. V.: Combining Static and Dynamic Views for architecture reconstruction. Proc. of the Sixth European Conference on Software Maintenance and Reengineering, Budapest, March 2002
- [Robak 2003] Robak S.: Modeling Variability for Software Product Families. osition Paper. In: M. Riebisch, J. O. Coplien, D. Streitferdt (Eds.): Modelling Variability for Object-Oriented Product Lines. BookOnDemand Publ. Co., Norderstedt, 2003. pp. 32-39.
- [Robbins et al. 1996] Robbins, J., Hilbert, D., Redmiles, D.: Extending Design Environments to Software Architecture Design. Proceedings of the 11th Annual Knowledge-Based Software Engineering (KBSE-96) Conference (Syracuse, NY), IEEE Computer Society, Los Alamitos, CA, September 1996
- [Royce 1970] Royce W.: Managing the Development of Large Software Systems. 1970 WESCON Technical Papers, v. 14, Western Electronic Show and Convention, Los Angeles, Aug. 25-28, 1970; Los Angeles: WESCON, 1970, pp. A/1-1 – A/1-9; Reprinted in Proceedings of the Ninth International Conference on Software Engineering, Pittsburgh, PA, USA, ACM Press, 1989, pp. 328–338.
- [Rugaber et al. 2000] Rugaber, S.: The use of domain knowledge in program understanding. Annals of Software Engineering, 2000
- [Rumbaugh et al. 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling And Design, Schenectady, New York, Prentice Hall, 1991.
- [RUP] Rational Unified Process, <http://www.rational.com/products/rup>
- [Salton McGill 1983] Salton, G.; McGill, M. J.: Introduction to Modern Information Retrieval, McGraw-Hill, 1983.
- [SEIATA] The Architecture Tradeoff Analysis Method (ATAM), <http://www.sei.cmu.edu/ata/ata-method.html>
- [SEIDALI] The Dali Architecture Reconstruction Workbench, <http://www.sei.cmu.edu/ata/products-services/dali.html>
- [SEIDE] Domain Engineering: A Model-Based Approach, <http://www.sei.cmu.edu/domain-engineering/domain-engineering.html>
- [Simon et al. 2001] Simon, F.; Steinbrekner, F.; Lewerentz, C.: Metrics based refactoring. In Proc. of European Conf. Software Maintenance and Reengineering, 2001, pp. 30-38.
- [Simos et al. 1996] Simos M., Creps D., Klinger C., Levine L., Allemang D.: Organization Domain Modeling (ODM) Guidebook, Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, June 14, 1996, <http://www.organon.com>
- [Schmidt 2000] Schmidt D.: Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects. John Wiley Sons; 1 edition (September 14, 2000)

- [Sommerville et al. 1997] Sommerville, I.; Sawyer, P., Requirements Engineering. John Wiley and Sons Ltd.
- [Sommerville 2001] Sommerville I.: Software Engineering. Sixth Edition, Addison Wesley, 2001
- [Standish 1995] Standish Group (1995) CHAOS report. www.pm2go.com, CHAOS chronicles.
- [Storey et al. 1997] Storey, M. D., Fracchia F.D., Mller H. A.: Rigi: A Visualization Environment for Reverse Engineering. Proceedings of the International Conference on Software Engineering (ICSE'97), Boston, U.S.A., May 17-23, 1997.
- [Suny et al. 2001] Suny, G.; Pollet, D.; LeTraon, Y. Jzquel, J.-M.: Refactoring UML models. In Proc. UML2001, LNCS 2185, Springer, 2001, pp. 134-138.
- [Svetinovic 2003] Svetinovic D.: Software Evolution: A Definition, <http://www.davors.com/se/evolution/definition.htm> (2003 viewed)
- [Tilley et al. 1998] Tilley, S. R.: A Reverse-Engineering Environment Framework. Software Engineering Institute, Carnegie Mellon University. (CMU/SEI-98-TR-005), 1998
- [UML] Unified Modeling Language (UML), version 1.5.
<http://www.omg.org/technology/documents/formal/uml.htm>
- [Vici et al. 1998] Vici A., Argentieri N., Mansour A., dAlessandro M., Favaro J.: FODAcOm: An Experience with Domain Analysis in the Italian Telecom Industry. In Proceedings of the Fifth International Conference on Software Reuse (Victoria, Canada, June 1998), pp. 166-175, IEEE Computer Society Press, 1998, see <http://www.intecs.it>
- [Wartik et al. 1992] Wartik S., Prieto-Daz R.: Criteria for Comparing Domain Analysis Approaches. In International Journal of Software Engineering and Knowledge Engineering, vol. 2, no. 3, September 1992, pp. 403-431
- [Weiss et al. 1999] Weiss D., Lai C.: Software Product Line Engineering: A Family Based Software Development Process. Addison Wesley, 1999
- [XP] Extreme Programming: A gentle introduction, <http://www.extremeprogramming.org>