

CASCADED REFACTORING FOR FRAMEWORK DEVELOPMENT AND EVOLUTION

LUGANG XU

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

DECEMBER 2005
© LUGANG XU, 2006



Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 978-0-494-16287-3

Our file Notre référence

ISBN: 978-0-494-16287-3

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

Abstract

Cascaded refactoring for framework development and evolution

Lugang Xu, Ph.D.

Concordia University, 2006

This thesis addresses three problems of framework development and evolution: identification and realization of variability, framework evolution, and framework documentation. A solution, called the cascaded refactoring methodology, is proposed. The methodology is validated by a case study, the Know-It-All framework for relational Database Management Systems.

The cascaded refactoring methodology views framework development as framework evolution, which consists of framework refactoring followed by framework extension. A framework is specified by a set of models: feature model, use case model, architectural model, design model, and source code. Framework refactoring is achieved by a set of refactorings cascaded from the feature model, to use case model, architectural model, design model, and source code. The constraints of refactorings on a model are derived from the refactorings performed on its previous model. Alignment maps are defined to maintain the traceability amongst the models.

The thesis broadens the refactoring concept from the design and source code level to include the feature model, use case model, and architectural model. Metamodels and refactorings are defined for the feature model and architectural model. A document template is proposed to document the framework refactoring.

Acknowledgements

First and most importantly, I am deeply indebted to my supervisor, Dr. Greg Butler. This thesis would not have come about without his persistent guidance, support, encouragement, and his faith in my aptitude throughout my Ph.D. program. He had not only been an academic supervisor in providing constant direction and focus to my research, but also a mentor — a role model who had affected me so much in the view of the world — with his kindness, open-mindedness and warmth. Thank you, Dr. Butler!

I would also thank the other members of my committee, for their feedback and input on this thesis.

I would like to thank our graduate secretary, Halina, TA program assistant, Pauline, and the secretary to chairman, Stephanie, for their tireless efforts in supporting me with their kind attentiveness and for providing me with ample opportunities to prove my teaching skills throughout this academic career.

I am thankful to Yue Wang for voluntarily and patiently proofreading parts of this thesis, and for discussion about the work and the way to present it. Several improvements of the writing are due to her comments.

I wish to especially thank Sue for her love and support throughout the entire process of my research and for having provided the much-needed motivation with her constant reminder of how long I have been in the program.

I would express my warmest gratitude to my parents and my grandma for loving me for whatever I am.

Most importantly, I thank God for granting me the talents and opportunities to achieve this accomplishment.

Contents

List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 The Problem	1
1.1.1 Object Oriented Application Framework	2
1.1.2 Framework Development Methodologies	4
1.1.3 Three Support Issues	7
1.2 Cascaded Refactoring Methodology	9
1.3 Case Study	11
1.4 Contributions	13
1.5 Thesis Organization	15
2 Background	16
2.1 Software Evolution	16
2.1.1 Maintenance and Evolution	17
2.1.2 Dimensions of Evolution	17
2.1.3 Laws of Software Evolution	18
2.1.4 Tools and Techniques	20
2.1.5 Software Process Models	21
2.2 Domain Specific Software Development	24
2.2.1 Software Reuse	25
2.2.2 Domain Engineering	25
2.2.3 Software Product Line	27
2.2.4 Other Work	28

2.3	UML Design Models	29
2.3.1	Unified Modeling Language	29
2.3.2	Structural Models	30
2.3.3	Behaviour Models	32
2.3.4	Model Extension	34
2.4	Feature Model	36
2.4.1	Feature Oriented Domain Analysis	36
2.4.2	Feature Oriented Reuse Method	38
2.4.3	Other Work	40
2.5	Use Case Model	41
2.5.1	Use Case in UML	41
2.5.2	Other Work	43
2.6	Software Architectural Model	45
2.6.1	Architecture Model	45
2.6.2	“4+1” View	46
2.6.3	Applied Software Architecture	47
2.7	Design Pattern	48
2.7.1	Pattern	49
2.7.2	Object Oriented Design Pattern	50
2.8	Framework	51
2.8.1	Framework Introduction	51
2.8.2	Framework Development	54
2.8.3	Framework Evolution	60
2.8.4	Framework Documentation	62
2.9	Refactoring	65
2.9.1	Program Refactoring	66
2.9.2	Refactoring Formalisms	71
2.9.3	Tool Support	74
2.9.4	Other Refactorings	75
2.9.5	Open Issues	77
2.10	Software Traceability	78
2.10.1	Requirements Traceability	79
2.10.2	Dimensions of Traceability	81

2.10.3	Traceability Characterization	82
2.10.4	Automation and Tool Support	84
3	Cascaded Refactoring Methodology	86
3.1	Models	87
3.1.1	Choice of Models	88
3.1.2	Feature Model	91
3.1.3	Use Case Model	96
3.1.4	Architectural Model	100
3.1.5	Design Model	105
3.1.6	Source Code	107
3.1.7	Model Notation	107
3.2	Alignment Maps between Models	113
3.2.1	Modeling Commonality and Variability	114
3.2.2	Maps	115
3.3	Cascaded Refactoring Methodology	130
3.3.1	Cascade of Refactorings	131
3.3.2	Model Refactoring	134
3.3.3	Documenting Refactoring	136
3.3.4	Refactorings	138
3.3.5	Conclusion	146
4	Know-It-All Case Study	148
4.1	Case Study	149
4.1.1	Introduction to the Domain	149
4.1.2	Introduction to the Framework	152
4.2	Case Study Models	153
4.2.1	Feature Model	153
4.2.2	Use Case Model	157
4.2.3	Architectural Model	158
4.2.4	Design Model	162
4.2.5	Source Code Model	170
4.3	Model Alignment Maps	171
4.3.1	Capability Feature Model to Use Case Model	172

4.3.2	Operating Environment Feature Model to Architectural Model	173
4.3.3	Domain Technology Feature Model to Design Model	174
4.3.4	Implementation Technique Feature Model to Source Code . . .	177
4.3.5	Use Case Model to Architectural Model	177
4.3.6	Use Case Model to Design Model	180
4.3.7	Architectural Model to Design Model	183
4.4	Model Refactorings	187
4.4.1	Example 1	187
4.4.2	Example 2	195
4.5	Discussion	205
5	Conclusion	209
5.1	Overview	209
5.2	Contributions	211
5.3	Limitations	213
5.4	Related Work	215
5.4.1	Framework Development	215
5.4.2	Refactoring	217
5.5	Validation Issue	220
5.5.1	Validation in Academic Refactoring Community	220
5.5.2	Ideal Industry Validation	222
5.6	Future work	224

List of Figures

1	Layer Structure	12
2	The Waterfall Model	22
3	A Class Diagram Example	32
4	A Collaboration Diagram Example	33
5	A Statechart Diagram Example	34
6	An Activity Diagram Example	35
7	An Example of UML Extensibility	35
8	A Feature Diagram of the Database Domain	37
9	Use Case Example	43
10	The Four Views of Applied Software Architecture	48
11	The Module View Metamodel	49
12	The Graphical Traceability Web	82
13	Feature Model Metamodel	92
14	Feature Model Metamodel Subset	96
15	Use Case Metamodel	97
16	Use Case Model Metamodel Subset	99
17	Architecture Metamodel	100
18	An Example of the UML Framework Concept	104
19	Feature Model Notation	108
20	Use Case Model Notation	109
21	Architectural Model Element Notation	110
22	Architectural Model Relationship Notation	111
23	Design Model Element Notation	112
24	Design Model Relationship Notation	113
25	Trace Maps	117
26	A Decision Record Example	137

27	A Simplified Relational DBMS Architecture	150
28	Know-It-All FORM Feature Model	154
29	Know-It-All Use Case Model	158
30	Know-It-All Architectural Model	159
31	Know-It-All Design Model: High Level View	163
32	Know-It-All Design Model I: Logical Layer	164
33	Know-It-All Design Model II: Physical Layer	166
34	Know-It-All Design Model III: DBMS Subsystem	168
35	The Sequence Diagram of Query Processing	169
36	Roadmap of Refactorings in Example One	188
37	Feature Model Refactoring: Decision Record 1	189
38	Modified Know-It-All Feature Model	191
39	Use Case Model Refactoring: Decision Record 1	192
40	Use Case Model Refactoring: Decision Record 2	192
41	Modified Know-It-All Use Case Model	192
42	Design Model Refactoring: Decision Record 1	194
43	Design Model Refactoring: Decision Record 2	194
44	Modified Design Model	195
45	Roadmap of Refactorings in Example Two	196
46	Architectural Model Refactoring: Decision Record 1	197
47	Architectural Refactoring I	198
48	Architectural Model Refactoring: Decision Record 2	198
49	Architectural Refactoring II	199
50	Architectural Model Refactoring: Decision Record 3	199
51	Architectural Refactoring III	200
52	Design Model Refactoring: Decision Record 1	201
53	Design Model Refactoring I	202
54	Design Model Refactoring: Decision Record 2	202
55	Design Model Refactoring II	203
56	Design Model Refactoring III	204

List of Tables

1	Design Patterns for Variability	51
2	Commonality of Framework Development Approaches	60
3	The Summary of Feature Model Elements and Relationships	94
4	The Summary of Architectural Model Elements and Relationships . .	103
5	Design Model Elements	106
6	Design Model Relationships	107
7	Entity and Qualifier Map of T_{fu}	170
8	Relationship Map of T_{fu}	171
9	Entity and Qualifier Map of T_{fa}	172
10	Relationship Map of T_{fa}	173
11	Entity and Qualifier Map of T_{fd}	175
12	Relationship Map of T_{fd}	176
13	Entity and Qualifier Map of T_{ua}	178
14	Relationship Map of T_{ua}	179
15	Entity and Qualifier Map of T_{ud}	181
16	Relationship Map of T_{ud}	182
17	Entity and Qualifier Map of T_{ad} : Part I	184
18	Entity and Qualifier Map of T_{ad} : Part II	185
19	Relationship Map of T_{ad}	186

Chapter 1

Introduction

Be happy.
It's one way of being wise.
~Colette

1.1 The Problem

This thesis addresses three problems of framework development and evolution: identification and realization of the required variability of a framework; framework evolution; and framework documentation. We propose the cascaded refactoring methodology as a solution to address the problems. The methodology is validated by a case study, the **Know-It-All** framework for relational database management systems.

Software reuse employs artefacts from existing systems to build new ones in order to improve productivity, reliability, and maintainability, and to reduce cost and development time [TRAC88]. Early experience with software reuse was limited to reuse of program source code. Object-oriented programming offers reusability of code via its techniques such as inheritance and composition. Class libraries with intelligent browsers and application generators were developed to help in this process. The contemporary reuse techniques have shifted the focus from code reuse to design and architecture reuse because of the larger potential benefits [LI93]. Software product lines and application frameworks are two of the latest cutting-edge reuse techniques.

A *software product line* describes a family of related software products in a specific problem domain [WL99]. These products share a common, managed set of features

satisfying the specific needs of the domain, and are developed from a common set of core assets in a prescribed way. Those features form a reusable platform, which can be used to build products through extension with variable features that are specific to particular products. By using a software product line, application developers are able to focus on product specific issues rather than issues that are common to all products.

As with all software, software product lines or frameworks undergo extensive evolution, and there is the need to have methodologies that support their development and evolution [MB99]. Although evolution in software product lines or frameworks is more complex due to their higher level abstraction and interdependency, it has not been studied much by the research community [SB99].

The cascaded refactoring methodology views framework development as framework evolution, which consists of framework refactoring followed by framework extension. A framework is specified by a set of models: feature model, use case model, architectural model, design model, and source code. Framework refactoring is achieved by a set of refactorings cascaded from the feature model, to use case model, architectural model, design model, and source code. The constraints of refactorings on a model are derived from the refactorings performed on its previous model. Alignment maps are defined to maintain the traceability amongst the models.

The thesis broadens the refactoring concept from the design and source code level to include the feature model, use case model, and architectural model. Metamodels and refactorings are defined for the feature model and architectural model. A document template is proposed to document the framework refactoring.

Next, we will at first give a brief view of object oriented application frameworks. Part 2 introduces the existing framework development methodologies. The issues addressed by our work are summarized in the last part.

1.1.1 Object Oriented Application Framework

An *application framework* provides a generic design within a given domain and a reusable implementation of that design [JF88]. An *object-oriented application framework* presents its design and implementation through a set of abstract classes and their collaborations [BJ94]. The design of a framework fixes certain roles and responsibilities amongst the classes, as well as standard protocols for their collaboration.

Customizing a framework by subclassing the given abstract classes makes the development of individual application cost-effective. Frameworks are extensible and flexible so that new components can be built and easily fitted into the infrastructure. Typically, a framework is developed by expert designers who have deep knowledge of the application domain and long experiences of software design. Frameworks offer a concrete realization of a software product line [CN02].

In the context of a framework, variability between applications in a specific domain is represented as *hot spots*, which is a variable aspect of the domain with associated responsibilities [PG94]. A framework provides simple mechanisms to customize each hot spot that resides in the framework architecture to instantiate concrete applications. A hot spot may have many hooks within it. A *hook* is a place in a framework that can be adapted or extended in some way, such as by filling in parameters or creating subclasses, to provide application specific functionality [PREE94]. Hot spots are usually realized with design patterns [PREE99]. *Design patterns* describes a commonly recurring structure of communicating components that solves a general design problem within a particular context [GOF94]. They capture the intent behind a design by identifying objects, their collaborations, and the distribution of responsibilities. Design patterns present proven solutions for how to internally structure hot spots in a framework [SCHM97]. Frameworks make heavy use of design patterns in the design and documentation [RJ97] [FSJ99].

The first widely used framework was the Smalltalk-80 user interface framework, called the Model/View/Controller(MVC) [KRAS88], which was developed in late 1970's. MVC divides the user interface into three parts; *models*, an application object that is independent of the user interface; *views*, which manages a region of the display; and *controllers*, which converts user input events into operations on its model and view. MVC was followed by other GUI frameworks including ET++ from the University of Zurich [WGM88]. There are a number of large commercial frameworks such as Microsoft Foundation Class (MFC) [PROS99], Taligent [CP95], Java Abstract Window Toolkit (AWT) and its successor Swing [DFK04]. Frameworks can be built on other domains, such as Choices for operating systems [RUSS91], and MET++ for multimedia applications [ACKE96]. Frameworks do not need to restrict the implementation to object-oriented languages. The Genesis database system compiler is a framework for database management systems [BBG+89]. It is implemented in the C

language.

Johnson and Roberts [RJ97] observe that a framework always evolves through a number of levels of *maturity* from White-Box to Black-Box, as the level of ease in the framework customization increases. Application developers of a *White-Box* framework have to understand the internal structure of the framework, and provide the subclasses to the abstract classes in the framework. Whereas a *Black-Box* framework encompasses a fairly complete set of concrete subclasses for each of the abstract class, and the customization can be done by choosing the appropriate subclasses for class composition. The core structure of a Black-Box framework is transparent to the application developers. A framework evolves from White-Box to Black-Box as the framework developers increase their understanding of the domain [JF88].

A framework is not easy to understand at first use. The large learning curve faced by the first time users of a framework is a serious impediment to successfully reaping the benefits of reuse [BD99]. Good framework documentation assists application developers in framework reuse [JOHN92]. Documentation is a key step in framework development, since the ability to write clear documentation that explains how application developers should reuse the framework means that the concepts of the design are clear and that the steps required for customization have been clearly thought out [BCC+02]. Hence, documentation verifies that the framework is easy to use, and this is the overriding goal of framework design.

1.1.2 Framework Development Methodologies

Developing a framework is different from developing an individual application because a framework has to cover all relevant concepts in a domain, whereas an application is only concerned with the application requirements [BMMB00]. Thus, standard software development methodologies are not sufficient for developing object-oriented frameworks [PG94]. Moreover, framework evolution should be considered in framework development since all frameworks seem to mature from initial versions through to a stable platform [RJ97]. Framework evolution as a concept is broader than software evolution because frameworks also evolve along maturity levels.

A framework as all software evolves [FSJ99]. *Software evolution* is “the dynamic behaviour of programming systems as they are maintained and enhanced over their life times” [BL76]. Perry [PERR94] classifies the source of software evolution into

three interrelated dimensions: the ever-changing environment, the process, and the developers' experiences. Software must continually evolve to remain satisfactory in use [LR02]. Nonetheless, continuous evolution may reach a point that software becomes too complex to evolve cost-effectively. Parnas [PARN94] refers to this problem as “*software aging*” and argues that the aging problem occurs in all successful software products, because uncontrolled changes deteriorate software structures and the software artefacts become inconsistent. He suggests design for change and precise documentation to address the issue.

Although research has been conducted on software evolution for decades, the outcome is still far from expectation [TURS00]. Experiences from both academia and industry have shown that, existing software development processes still lack adequate support to deal with software evolution [LR02]. Mens [MENS05] observes a set of open issues in software evolution. He claims that evolution techniques for higher levels of abstraction other than source code evolution are expected.

Several methodologies have been suggested for the development of frameworks. However, the methodologies vary quite widely, and have been poorly supported by notations for models [FSJ99]. The existing methodologies can be classified into Bottom-Up, Top-Down, Hot Spot Generalization, and Use Case Driven. They are summarized as follows [FSJ99]:

- Bottom-Up approaches build a number of applications in the framework domain prior to developing the first version White-Box framework [JF88] [WW93]. The common and variable features of those applications are identified during the abstraction of the application design and encompassed in the framework [RJ97]. The framework is refactored in an iterative process until the framework can handle all applications in the domain. Each new version of the framework is validated via instantiating applications from the framework.
- Top-Down approaches perform analysis on the framework domain and capture all features in terms of commonality and variability [STAR96] [WL99]. The result of domain analysis is used to define the Domain Specific System Architecture (DSSA) and appropriate reusable components that can be customized during actual application development [KKL+98]. The DSSA can be instantiated to frameworks since frameworks are a kind of DSSA [TRAC94].

- Hot spot generalization approaches plan all applications before starting to build a framework. They use an application object model to capture the domain specific knowledge [SCHM97] [PREE99]. Variability in the model is identified as hot spots, which is realized by a hot spot subsystem. Each variation point in the object model is associated with a hot spot subsystem that provides the variability. Hence the framework class structure is “generalized” from the application class structure.
- A *use case* describes the external visible behaviour of a system [JCJO92]. Use case driven approaches start with domain analysis and organize the results into use cases. Variability is modeled in use cases using variation points or generalization [JGJ97] [DW98]. They then proceed with either top-down or hot spot generalization approaches. The variation points are realized with design patterns.

Design of a framework emphasizes the elicitation of the required variability. Use cases models can be employed to capture the requirements since use cases have become one of the standard techniques to model software requirements, especially after the emergence of the Unified Modeling Language (UML) [BRJ99]. However, non-functional requirements such as performance or implementation standard might be difficult to model in use cases, due to its intrinsic “function-oriented” property. Furthermore, experiences [GFA98] [VAM+98] have shown that readability of use case models may be decreased by incorporating variability into already complicated models.

Traceability is the ability to trace the dependent items within a model and the ability to trace the corresponding items in other related models [PB90]. It is used to know the exact relationship between each requirement and its corresponding design and implementation, also to verify whether the requirement is implemented [RG93]. Furthermore, framework development requires an iterative approach in which the framework is refined a number of times [BOOC94]. As mentioned earlier, a framework always evolves along two dimensions during its development lifecycle. Impact on the design and implementation due to a change in requirements should be clearly identified, propagated, and documented to maintain the traceability during evolution.

1.1.3 Three Support Issues

Based upon the above discussion, we observe that the existing framework development methodologies lack adequate support to three issues. They are summarized as follows [BX01]:

1. Identification and realization of the required variability of a framework

Johnson [JOHN93] claims that an ideal way to develop frameworks should be an iterative process which is composed of domain analysis, generic design, and validation. Domain analysis is a process to capture and represent information about a family of applications in a domain [CN02]. It is an indispensable step to obtain the requirements of a framework [RJ97]. Identification of the required variability is particularly important for framework development because a framework inherently contains more variability than a typical application. The bottom-up and hot spot generalization approaches do not define an explicit domain analysis stage, instead, they use standard use cases or object models to capture and organize the analysis result. Some of the use case driven approaches such as FeaturSEB [GFA98] and top-down approaches use a feature model to capture the variability, which has been a proven solution [KCH+90].

The common and variable aspects identified during the domain analysis should be reflected appropriately into framework design [SCHM97]. Keeping traceability from the requirements to design and implementation is essential to guarantee the realization of the required variability. Use case driven approaches such as Reuse-driven Software Engineering Business (RSEB) [JGJ97] place an emphasis on keeping the traceability links of the representation of variability amongst the models of a framework. Top-down approaches such as Feature Oriented Reuse Method (FORM) [KKL+98] also propose a guideline to map requirements to architecture and object models. Nonetheless, those guidelines are too general to ensure and validate the realization of the identified variability in framework design and implementation [FSJ99].

2. Framework evolution

Johnson and Foote [JF88] claim that developing a Black-Box framework at

the initial stages of the framework's history is extremely expensive and difficult. Thus, most frameworks start their lifecycle as a White-Box framework and evolve to a Black-Box framework in an iterative process. Framework design takes iteration because of three reasons: mistakes in requirements due to the complexity of domain analysis; mistakes in abstraction due to inadequate applications for generalization, which is very expensive; and the law of reuse, which requires reusable software to be used in a context other than its initial context to prove its reusability [FSJ99].

Only bottom-up approaches explicitly specify the common path a framework takes in evolution [FSJ99]. However, there is no explicit description of where and how a framework evolves when it has reached a certain state [MB99]. Refactoring is used extensively in bottom-up approaches to restructure the code and design of frameworks without changing the visible behaviour [OPDY92]. Nonetheless, the transformation is only limited to source code and design, and may cause inconsistency between different software artefacts during evolution [MENS05]. Furthermore, changes in a framework may cause conflicts between the framework and the existing applications developed with the framework, which is identified as one of the most common problems regarding framework evolution [CHSV97]. Another problem is the increased structure complexity, which causes a framework difficult to be comprehended and reused. Bosch et al. [BMMB00] states that these issues have not been addressed by the existing framework development methodologies.

3. Framework documentation

Different audiences are concerned with different aspects of framework documentation [BD99]. Good documentation has to suit different audiences to meet their needs. Two groups of audiences are application developers to reuse frameworks; and framework maintainers to evolve frameworks [BD99].

Framework design is made abstract to accommodate commonality; is sometimes incomplete in order to provide extensibility; and is complex because the collaboration and dependencies among classes can be indirect and obscure [JOHN92]. Thus, understanding a framework is more difficult than understanding an application. Nonetheless, a framework must not be too complex to be comprehended

efficiently from the reuse viewpoint [BCC+02]. Therefore, accurate and comprehensive documentation is essential to reuse frameworks from the perspective of application developers [FSJ99]. However, such a documentation approach does not exist [BD99].

Parnas [PARN94] argues that good documentation is crucial to deal with software aging. Keeping traceability during software evolution propagates the changes on requirements to design and implementation, so to improves software maintainability [DW98]. However, few existing development methodologies have provided a set of coherent models as the notion to document a framework, and to support preserving traceability with precise guidelines. Fayad et al. [FSJ99] state that framework documentation is still one of the open issues in the framework research area.

1.2 Cascaded Refactoring Methodology

Current programming practice usually involves maintaining and updating programs in source code form. Opdyke [OPDY92] categories software changes into three levels: high level requirement changes, low level source code changes, and the intermediate level between them. He introduces the term *refactoring* as “reorganization plans that support change at an intermediate level”. For example, the refactoring that moves a member variable from one class to another class. He also identifies the intrinsic property of refactorings: refactorings should not change the *behaviour* of a program. Opdyke uses *preconditions* to preserve behaviour during refactorings. Tokuda and Batory [TB99] follow his work and propose additional refactorings to support design patterns installation, i.e. change class structure to an appropriate design pattern to increase flexibility. In order to facilitate the iterative process of framework development, we take the idea of refactoring and extend the concept into other type of software artefacts.

We propose the *cascaded refactoring methodology* to address the earlier mentioned issues and provide a solution. The methodology is a hybrid approach, which combines the modeling aspects of the top-down approaches, and the iterative refactoring approaches of the bottom-up community. The methodology views framework development as framework evolution, which is *framework refactoring* followed by *framework*

extension. The methodology only focuses on framework refactoring at the current stage.

A framework is specified by a chosen set of models: a feature model that identifies and organizes the commonality and variability of the framework; a use case model that captures the requirements; an architectural model that specifies the high level design in term of layers and subsystems; a design model that illustrates the interactions of classes and objects; and source code. The required variability of a framework is identified and captured into the feature model. In order to describe refactorings and justify preservation of behaviour, metamodels of feature model and architectural model are defined. The use case metamodel proposed by Rui [RB03] is adopted in the methodology.

The methodology stresses traceability between the models. It defines a set of *alignment maps* between the models to specify the traceability links amongst them. An alignment map between two models maps every entity and relationship in the domain to an object in the range without altering the functionality and variability. The realization of the required variability of the framework is obtained by keeping the alignment maps from the feature model and use case model, to the architectural model, design model, and source code.

The process of *cascaded refactoring* is a series of refactorings of the models. The impact of the refactorings on a model M_i to a model M_j , is translated via the alignment maps that have M_i as the domain and M_j as the range. Refactorings of a framework is achieved through a set of cascaded refactorings on the models. Changes on the requirements of a framework during the framework evolution are propagated to the design and implementation by cascading the refactorings from the feature model to other models.

We have defined a set of refactorings for the feature model, the use case model, and the architectural model. The notion of the preserved “behaviour” of refactorings of those models is clarified. For each model, a partial list of refactorings with preconditions is defined.

The iterative process of framework evolution expects consistent and comprehensive documentation. *Rationale* of software design allows maintainers to follow the reasoning used by the designers [KEAN97]. The rationale behind refactorings of a

framework should be recorded. The methodology views a refactoring as an issue-driven activity. The overall rationale is a collection of *decisions*, which is organized into a template defined in the methodology. Each decision records the *rationale*, *choice*, *argument*, and *impact* of an individual refactoring on one of the models. The process of cascaded refactoring is documented by a series of decision record collections. As part of the framework documentation, the refactoring document presents a clear *roadmap* of the sequence of applied refactorings on the involved models, in order to evolve the framework.

The cascaded refactoring methodology broadens refactoring as a concept to feature model, use case model, and architectural model. It weaves together steps for partial domain engineering and steps of system refactoring. The methodology has been validated by a case study.

1.3 Case Study

The research on software methodology in an academic setting needs a concrete case study for the purpose of validation. Therefore, Know-It-All, a database management system (DBMS) framework has been underway since 1997 [BCC+02]. It is a case study to validate the cascaded refactoring methodology. In our initial ambition, Know-It-All is intended to support a variety of data models of data and knowledge, different paradigms integration, and heterogeneous databases. It is intended to be used to customize advanced database applications in bioinformatics. While the applications to bioinformatics allow the framework to be verified, along the way, the research in software technology leads to a platform for research in database technology, which in turns leads to advances in bioinformatics and genomics.

Know-It-All is designed with scientific databases in mind, and does not provide transactions. Instead, it provides a data feed mechanism for bulk or incremental data loads. The prime task is querying of existing data. The framework provides a generic infrastructure for database management systems and allows them to support a range of data models (relational, object, object-relational, etc) where the data model itself, and its constituents for query language, query optimizing, indexing, and storage have clearly defined roles.

A database in Know-It-All is seen as a series of layers, each of which provides

the common interface. The usual breakdown of responsibilities into physical, logical, conceptual, and view layers is followed by **Know-It-All**. Each *layer* in **Know-It-All** is basically a translator between its client layer and its supplier layer, as shown in Figure 1. A layer provides a mechanism to decompose or translate queries, and a mechanism to reconstruct answers (for example, an execution plan for relational algebra expressions). The translation is done with the aid of the schema, and produces both the translated query, and the mechanism to reconstruct answers. The layer architecture is adapted from one for heterogeneous databases [MB96], while the reconstruction is carried out through navigating an iterator tree which represents the execution plan. **Know-It-All** will eventually incorporate composite databases (such as integrated or heterogeneous databases) and make no distinction between simple and composite databases.

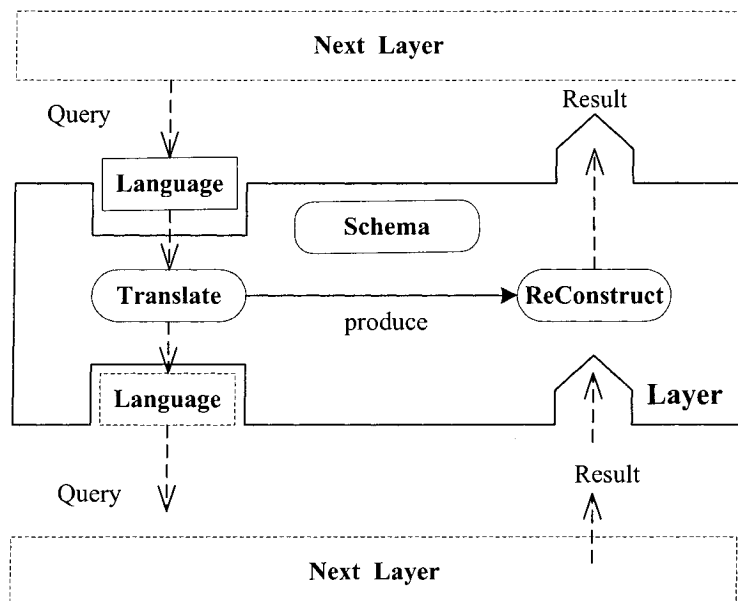


Figure 1: Layer Structure

Limited by time and resources, to date **Know-It-All** only implements a relational DBMS. The first version prototype is implemented with GNU C++, with some Java for the user interface, and XML for communication of data between the C++ framework and the Java tools. It supports query processing, data feed and schema definition. It also contains two sub-frameworks: OPT++ for query optimization [KD98]

and Gist for index technique [HKP97]. The prototype provides a generic infrastructure for relational database management systems and components for query optimizing, indexing, and storage management. ANSI SQL-92 is chosen as the query language, since it is the standard query language in the relational DBMS domain. Flat text files are used as a storage medium, as well as a conventional physical storage manager from PostgreSQL [POST01]

We construct the feature model, use case model, architectural model, design model, and implement source code of the **Know-It-All** framework. The feature model, use case model, and architectural model conform to the metamodels. The design model follows the UML standard. The alignment maps between the feature model, use case model, architectural model, and design model are specified. Based on the models and maps, we demonstrate the cascaded refactorings with two refactoring examples. All involved refactorings are recorded with the document template of refactoring.

1.4 Contributions

We have made the following contributions:

1. The concept of refactoring is extended to all models and not just source code and class design.

In terms of level of abstraction, research on object-oriented software refactorings has been mainly focused on source code and design level. The methodology extends the notion of refactoring to the feature model, the use case model, and the architecture. The invariants of refactorings on those models are identified and a partial set of refactorings is defined for each of those models. Rui [RB03] has extended the use case model refactorings.

2. The cascaded refactoring methodology is proposed.

Framework evolution can be viewed as framework refactoring followed by framework extension. The methodology addresses three issues in framework development (see page 7). It places an emphasis on traceability of the models of a

framework, in order to realize the required variability, and to cascade refactorings to maintain the consistency between the models during framework evolution. Refactorings of a framework is achieved with a set of cascaded refactorings on the models. Framework documentation also benefits from using the models to specify a framework, using alignment maps to specify the traceability links, and using documented refactorings to specify the evolution steps.

3. A set of models are chosen for framework development and evolution.

Precise guidelines to specify a framework with models are not given by the existing methodologies. We choose a set of coherent models to specify a framework across the analysis, design, and implementation. The models are able to express the commonality and variability of a framework. Metamodels are defined to exactly specify the models, to clearly describe the refactorings, and to allow the behaviour preservation of refactorings to be justified.

4. Alignment maps are defined to maintain the traceability of the models.

A set of alignment maps are defined to specify and maintain the traceability links between the models. The maps provide a precise guideline to aid the transition between different models. Change propagation during framework evolution is obtained by keeping the alignment maps between the models.

5. A document template is defined to record the refactorings.

The methodology incorporates the issue-driven approach and views a refactoring as an issue-driven activity. The overall refactoring rationale is a collection of decisions, which is documented with a template. Each decision records the intent, choice, arguments, and the consequences of a refactoring. The refactorings on the models are recorded in the cascading sequence, as part of the framework documentation, which is helpful to the design and maintenance of the framework.

6. An academic setting framework for relational database management systems, called **Know-It-All**, is developed as the case study to validate the methodology.

The methodology has been validated with the **Know-It-All** case study. The models of **Know-It-All**, and the alignment maps between the models are specified and validated. Sample cascaded refactorings on the framework are demonstrated.

The limitations on our research are:

1. Cascaded refactoring is not a complete methodology.

The methodology aims to cover framework evolution. However, our work only focuses on the framework refactoring. The metamodels are not defined in formal languages. We only have initial treatment of variability within framework models. The alignment maps are not full maps and are only defined on the subset of each model. Source code related alignment maps are not defined. The invariants of different models during the cascaded refactorings mainly stress the functionality. More quality attributes should be considered in the context of a framework. We have only defined a small subset of refactorings for the models.

2. The architectural modeling is rather limited.

We had difficulty to choose an appropriate architectural model for frameworks. The traditional way of subsystem and interfaces is used to present the high-level framework architecture. Other views of architecture such as the process view, and deployment view are not covered.

3. The case study is not big enough.

Know-It-All only supports a small set of DBMS features. The design still lacks flexibility. The indexing techniques are quite limited. More work is needed on the physical layer. We only did a small number of refactorings on the framework. The framework has not been validated with “rule of thumb”, i.e. a framework should be verified by building a number of applications from the framework in different contexts [RJ97].

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 introduces the background knowledge. Chapter 3 elaborates the cascaded refactoring methodology. The Know-It-All case study is given in chapter 4. The final conclusion and future work is discussed in chapter 5.

We assume the reader is familiar with the C++ programming language [STRO97]. The *Unified Modeling Language* (UML) [RATU03] will be introduced in chapter 2.

Chapter 2

Background

Express yourself completely,
then keep quiet.
~Lao Tzu

This chapter introduces the fundamental background to understand the problem and the solution that are described in this dissertation. Section 1 introduces software evolution. Section 2 gives an overview of domain engineering and software product lines. Section 3 introduces the concepts and notations of UML design models. Section 4 presents feature models with two seminal work FODA and FORM. Section 5 introduces the basic concepts and notations of use case models. Section 6 reviews software architectural models. Section 7 introduces object-oriented design patterns. Section 8 introduces object oriented application frameworks. Section 9 discusses software refactoring. Software traceability is introduced in the last section.

2.1 Software Evolution

Software systems evolve to meet changing requirements. In this section, we introduce the basic concepts relating to software evolution. The first part gives a brief view on software maintenance and evolution. Part 2 explains the reason why evolution is an intrinsic property of software systems. The laws of software evolution are described in part 3. Part 4 discusses evolution from the perspective of tools and techniques. The last part introduces software process models and the open issues for software evolution.

2.1.1 Maintenance and Evolution

Software is much different from the products of other engineering fields in many ways. It is intangible, complex, and very difficult to make correct changes [SUMM00]. Hence, the approaches from existing engineering profession cannot be directly applied on software development. *Software engineering* is the process of solving clients' problems by the systematic development and evolution of large, high-quality software systems within cost, time, and other constraints [LL01]. The term "software engineering" was proposed at a NATO conference at 1968 [NR68].

Typically, software engineering work is organized into *projects*, which can be divided into three types: modifying an existing system, developing a new system from scratch, and building a new system from existing components [LL01]. Most software projects are of the first type, which is generally accepted as "*software maintenance*". It is a process not only including bug fixes, but also constant changes requested from the customers. Swanson [SWAN76] proposed one of the first typologies for software maintenance activities. The maintenance types are classified into *corrective*, *adaptive*, and *perfective*. The classification was adopted by the IEEE as the standard for software maintenance [IEEE93]. Industry experiences have shown that software maintenance activities span the production life of a software system and can account for as much as 80% of its total budget [RBCM91]. After several years of changes, software systems are often significantly larger and different with their original state. Thus, the word "evolution" is used to describe the process over software lifecycles.

Software evolution is "the dynamic behaviour of programming systems as they are maintained and enhanced over their life times" [BL76]. It is a process of gradual changes that takes place in a software system over a period of time. It has been generally accepted that, software must be continually adapted, enhanced, and extended if it is to remain satisfactory in use [LR02]. Software evolution is usually classified into three categories: *corrections* in various artefacts that cover from requirements, to design and code; *improvements* on different quality factors, and *enhancements* on general functionality and features [PERR94].

2.1.2 Dimensions of Evolution

Given a software system, Perry [PERR94] states that evolution is an intrinsic property and observes three interrelated dimensions of sources of evolution:

1. *Domain* is divided into three parts. First, the real world and the set of observations of the real world from the system perspective, that is, the system context. They are the fundamental sources of system evolution because they inherently evolve themselves. Second, the system specification that is based on the abstraction of the context. Third, the theories that transform the specification through different abstraction levels into an operational system.
2. *Experience* is the basis of the good judgement used in the process of abstraction and reification of the system. Experience is gained from feedback, experiments, and accumulated knowledge from building, evolving and using the system.
3. *Process* consists of three components: methods that are based on the theory and experience; technologies that support automation of various aspects of the building and evolving process of the system; and the organizational environment in terms of culture and standards for systems and processes.

Perry's result indicates that software must evolve to meet new requirements. However, continuous evolution may reach a point that new releases increase the complexity and decrease the maintainability of software. It becomes too complex to evolve cost-effectively. Parnas [PARN94] refers to this problem as “*software aging*” and argues that the aging problem occur in all successful software products due to the failure of the product's owners to modify software, and bad quality changes that deteriorate software structures. He suggests use design principles that expect changes, keep change encapsulated, and document changes precisely and completely. However, design for change is difficult since it implies predicting the future.

2.1.3 Laws of Software Evolution

The most prominent studies of software evolution have been directed by Lehman and his colleagues over a thirty year period dating back to the mid-1970's [LB85] [LEHM96] [LR98] [LR01]. The case study covers several large scale setting projects, including the IBM OS/360 operating system, FEAST (Feedback, Evolution And Software Technology) and its successor FEAST/1 and ongoing FEAST/2 [LR03]. They have formulated eight *laws* of software evolution and claims that these laws reflect observable behaviour that is “organisational and sociological in nature” and irrelevant to

the technology being applied in the evolution process. The eight laws are summarized in the following list:

1. Continuing change: a software application must be continually adapted to satisfy progressively emerging changes in its operational environment.
2. Increasing complexity: software complexity increases due to the uncontrolled adaptation activity.
3. Self regulation: evolution processes are self regulating within large organisations.
4. Conservation of organisational stability: the average productivity rate on an evolving system is steady over its life cycle.
5. Conservation of familiarity: the average knowledge and skills of all participants to evolve a system are constant during evolution.
6. Continuing growth: the functionality and features of a software application must be continually increased to maintain user satisfaction. The situation is often caused by fulfilling requirements that were skipped in the previous releases due to prioritized trade-off decisions.
7. Declining quality: quality of software applications decline as they evolve.
8. Feedback system: evolution processes are composed of multi-level, multi-loop, multi-agent feedback systems. This law addresses evolution that involves the use of Commercial Off The Shelf (COTS) software. A COTS application may have to be changed due to the new requirements from a larger system in which it is integrated.

Many empirical studies follow Lehman's work. Kafura and Reddy [KR87] analyzes the relationship between software complexity metrics and maintenance. They propose two types of complexity metrics, *code metrics* and *structure metrics*, which were used to quantify the complexity of different releases of a software system. They claim that both types of complexity increase during the system maintenance, and extreme changes of complexity in a procedure or module may suggest possible errors during the maintenance activity. Their result conforms to Lehman's fourth law.

Gall et al. [GJKT97] examine the structure of a telecommunication switching system (TSS) based on the data stored in a product releases database (PRDB). The system is viewed as a hierarchy of four different levels: *system*, *subsystem*, *module*, and *program*. They report that although the growth of the system conforms to Lehman's laws at the system level, individual subsystems and modules often do not, and exhibit significant upward or downward fluctuation in their size across almost all releases.

2.1.4 Tools and Techniques

Lehman's laws and other related work focus on the nature of software evolution and the properties of the evolution phenomenon. Other research focuses on the development of methods and tools that are used to facilitate software evolution. Mens et al. [MBZR03] proposes a taxonomy to categorize the tools and techniques for software evolution in order to find the way to improve and collaborate the tools. The categorization is summarized as follows:

1. *When* to integrate changes into a system? This category is divided into three subgroups: *time of change*, whether the changes occur at compile-time, load-time, or run-time; *change history*, which often relies on version control tools; and *change frequency*, which is concern with the interval of changes during software evolution.
2. *Where* does a change is made? This category is divided into four subgroups: *artefact*, which ranges from requirements through architecture, design, source code, test cases and documentation; *granularity*, which refers to the coverage of artefacts to be changed; *impact*, which is concern with the consequence of changes; and *change propagation*, which focuses on the way to keep traceability between the artefacts (see Section 2.10 for detail on traceability).
3. *What* are the changed system properties? This category is divided into four subgroups: *availability*, indicates whether the system can be stopped for changes; *activeness*, whether the system can automatically make necessary changes by itself; *openness*, whether a system is specifically built for extension such as a framework; and *safety*, whether the behaviour of a system can be preserved by changes at compile-time or run-time.

4. *How to support changes?* This category is divided into four subgroups: *degree of automation*, whether changes are performed automatically such as refactoring tools; *degree of formality*, whether a tool is based on mathematical formalism; *process support*, whether a tool can be efficiently integrated into development processes; and *change type*, whether a change being made modify the structure or behaviour of a system.

Cook et al. [CJH00] define *evolvability* as the “capability of software products to be evolved to continue to serve its customer in a cost effective way”. As a software quality factor, evolvability is determined by *analysability*, the ability to understand a system and its necessary changes; *changeability*, the ability to transform a system from one stable state to another state; *stability*, the flexibility to allow necessary changes without altering structure or existing functionality; *testability*, the ability to verify functional and non-functional requirements; and *compliance*, the ability of a software product to adhere to standards relating to maintenance. They propose an evolvability metrics model based on the classification and addresses evolution of a system from three areas: software product quality, software evolution processes, and the organisational environment. They claim that their work integrates modeling techniques into the evolution phenomenon, hence includes the both views of evolution.

2.1.5 Software Process Models

Software process models are general approaches for organizing a software project into activities in sequences, to be followed by software developers to perform the work [BOEH88]. They are mainly used to “determine the order of the *stages* involved in software development and evolution and to establish the transition criteria for progressing from one stage to the next”. A *software development methodology* is a critical tool to manage software development processes in terms of risk control, software design, quality assurance, cost estimation, etc, to meet the clients’ requirements [TRUS99]. A mature methodology increases the chances of making good quality products “by decreasing the overall complexity of the software engineering effort” [BERA93]. Thus, a process model is different to a software methodology because a methodology emphasizes on the stage transition and products representation.

Right after the appearance of “software engineering”, Royce [ROYC70] proposed the classic stage-wise waterfall process model, which has become the basis for most

software acquisition standards [BOEH88]. Software systems are developed in successive stages and the output of the previous stage is the input of the following stage as a cascade. Thus, the model is called waterfall model, and each stage is also referred as “*phase*”. Over the past thirty years since its emergence, some of its initial difficulties have been addressed by adding extensions. Figure 2 [PREE92] shows an extended waterfall model which stresses quality assurance. The activities of each phase are described as follows:

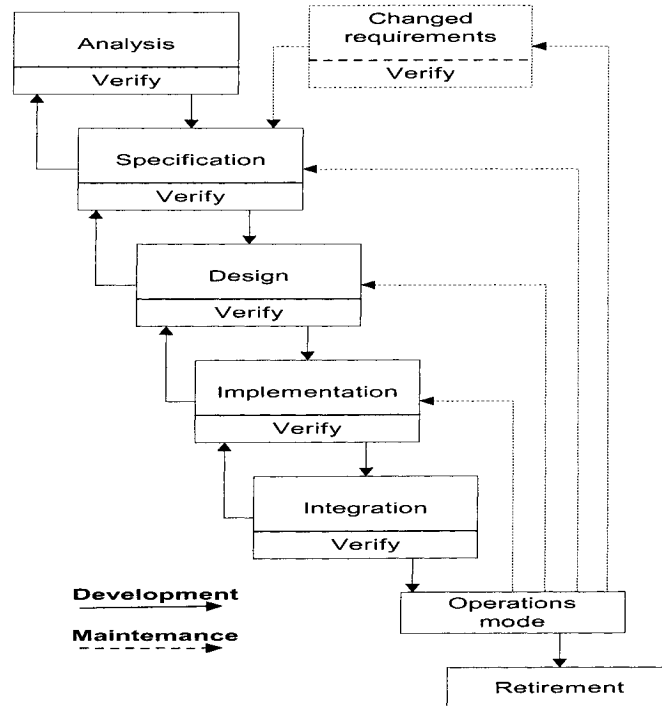


Figure 2: The Waterfall Model

- *Analysis* establishes the system’s requirements by consultation with system users.
- *Specification* defines the requirements completely and precisely in a prescribed format.
- *Design* finds a solution to satisfy the software requirements.
- *Implementation* realizes the design as a set of programs or program units.

- *Integration* combines the individual program units or programs together and test as a complete system.
- *Operations Mode* delivers the system to the clients and put into practical use. Maintenance work might be necessary at this stage to deal with changed requirements.
- *Retirement* ceases the use of the system from the clients' organization.

Verification has to be performed at the end of each phase to ensure the phase products conform to the expected result or standards. The development process can navigate through phases backward and forward to propagate changes to keep traceability.

Based on the waterfall model, many other process models have been invented such as evolutionary model and spiral model [BOEH88]. The recent eXtreme Programming (XP) process model is a lightweight approach of software development [BECK99]. XP delivers a software system to its customers as early as possible and implements the changes according to the feedback thereafter. It focuses on software code quality and tests, quickly responds to the suggested changes with refactoring.

Rajlich and Bennett[RB00] state that the existing software processes do not provide adequate support for software evolution. They propose a “staged model” that emphasizes maintenance activities. Software life cycle is viewed as five stages:

1. *Initial development*: the first version of software is developed and released to the clients to be put into practical use. Team expertise and familiarity to system architecture should be obtained to support the work at the later stages.
2. *Evolution*: the iterative process of evolving software based on the feedback from the clients to satisfy new requirements.
3. *Servicing*: software enters into this stage when it starts “aging”, mainly caused by inadequate expertise and familiarity which should be prepared in the initial stage. Hence, only minor defect repairs and simple functional changes are applied.
4. *Phaseout*: when servicing cannot be performed cost effectively any longer, clients try to gain benefits from the unchanged software as long as possible, and often have to work around deficiencies.

5. *Closedown*: the clients shut down the software system and possibly replaces it with a new system.

The global anxiety of “Y2K” problem before year 2000 has raised the awareness of how extensively software evolution efforts span a system’s productive life and how important evolution is. However, experiences from both academia and industry have shown that, existing software development processes still lack support to deal with software evolution in a cost-effective way [LR02] [MENS05]. Turski [TURS00] argues that the problem of adapting existing software to evolving specifications remains largely unsolved, perhaps is algorithmically insoluble in full generality. Although research has been conducted on software evolution for decades, the outcome is still far to expectation. Mens [MENS05] observes the following open issues in this area:

- Formal methods that stress evolution as an essential fact.
- Evolution techniques that target at higher level of abstraction other than source code evolution.
- Development processes that support evolution from both technological and managerial perspectives.
- Empirical studies on software evolution with large industrial settings.
- Quality factors that should be preserved during evolution to avoid software aging.

2.2 Domain Specific Software Development

Early experience with software reuse was limited to reuse of program code, data structures, and class libraries in the new software projects [PRIE89]. Recent reuse research concentrates on the development of common architecture with highly reusable components for closely related applications in a domain [JGJ97]. In this section, we will at first give an overview on software reuse. Part 2 introduces domain engineering. Part 3 discusses software product lines. The last part presents some famous domain engineering methods.

2.2.1 Software Reuse

Reuse is the use of previously acquired concepts or objects in a new situation, it involves encoding development information at different levels of abstraction, storing the representation for future reference, matching of new and old situations, duplication of already developed objects and actions, and their adaptation to suit new requirements [PRIE89]. *Software reuse* involves the use of artefacts from existing systems to build new ones in order to improve productivity, reliability, and maintainability to reduce cost and development time [TRAC88].

Reusable software products, which are easy to be changed to adapt future requirements, are a way to reduce development costs [CN02]. Meanwhile, feedback, debugging, and experience gained through reuse improve the quality of products in an iterative manner. These benefits have been a strong driving force of software engineering method research and development for a long time [JGJ97].

Early experience with software reuse was limited to reuse of program source code [PRIE89]. As a consequence, programming languages were developed to support code reuse, such as parameterization, data sharing via data types, code blocks, information hiding, modules, generic packages, objects and classes, etc. Code reuse was further supported by the wide application of software libraries, such as the C++ Standard Template Library (STL) [STRO97], which improve software development productivity, and have been practiced in nearly every commercial organization [LI93]. The contemporary reuse techniques have shifted the focus from code reuse to design and architecture reuse because of greater potential benefits [LI93]. One of the latest cutting-edge reuse techniques is the domain engineering and software product-line. It is based upon the idea that reusability depends upon a context of the problem and its solution, which themselves are relatively cohesive and stable [ARAN94].

2.2.2 Domain Engineering

A domain can be considered from two perspectives: either the target world that an individual application addresses, or a set of applications [SIMO97]. The latter alternative is used in domain engineering. Tracz [TRAC94] defines a *domain* as a collection of problems and a collection of existing or future applications perceived to be similar. It is an area of knowledge that includes a set of concepts and terminology understood by practitioners, and the way how to build software systems in that

area [ARAN94]. *Domain engineering* is a systematic design process of architecture and a set of reusable assets (components and other work-products) that can be used to construct a family of related applications in a given domain [CE00]. It incorporates business criteria and produces supporting rationale, models, and architectures to make good decisions. It plays a key role in providing a stable architecture and components for reuse.

Domain engineering is usually divided into three phases: domain analysis, domain design, and domain implementation [CE00]. Domain engineering begins with *domain analysis*, a process “for capturing and representing information about applications in a domain, specifically common characteristics and reasons for variability” [CN02]. *Domain scope* is the set of selected target applications. The domain scope should be clarified before starting the domain analysis. Domain analysis investigates both the *problem domain*, the context and requirements, and the *solution domain*, i.e. the applications [SCHM00]. The input for domain analysis includes example systems, user requirements, domain expertise, and future trends that can be gathered from customer surveys, consultation with experts, and projections of future requirements. The input is analyzed to identify and characterize elements that are common to all family members, i.e. *commonality*, and to deal with elements that vary between family members, i.e. *variability* [CE00]. Domain analysis includes three steps:

1. Domain identification and scoping: investigate all the applications in the domain and try to find the reusable parts in the applications
2. Selection and analysis of examples, requirements, and future trends: the reusable components have to reflect possible future requirements
3. Identification, factoring and clustering of feature sets: analysis models are used to gather features into a decision framework. The domain terminology is accumulated.

The output produced by domain analysis includes the taxonomy (glossary or data dictionary) of the concepts in the problem domain and solution domain, and the *Domain Software Specific Architecture* (DSSA) across applications in the domain [TRAC94]. A DSSA provides an infrastructure to describe the essential characteristics of the application family, appropriate features for specified customers, and the process how to refine it [TRAC94].

Domain design produces the core architecture and reusable components for a family of applications. Typically, the design involves the selection of architectural styles [SG96]. Variability between applications should be provided by the core architecture. Other than the architecture itself, a production plan is also provided to specify the guidelines of building applications from the architecture [CE00]. The architecture and components are implemented during the *domain implementation* phase.

2.2.3 Software Product Line

A *software product line* describes a family of related software products in a specific problem domain [CHW98]. These products share a common, managed set of features satisfying the specific needs of the domain, and are developed from a common set of core assets in a prescribed way. The features form a reusable platform, which can be used to build products through extension with variable features that are specific to particular products. By using a software product line, software developers are able to focus on product specific issues rather than issue that are common to all products. Large companies, such as Hewlett-Packard, Nokia, and Nortel, have found that a product line approach of software development can yield remarkable quantitative improvements in productivity, time to market, product quality, reusability, and customer satisfaction [CN02].

Typically, there are two relatively independent development lifecycles in software product lines: one for the software product line itself; the other is for each product instantiation. The *product instantiation* is the process of creating a specific software product using a software product line [CHW98].

Developing a product line is not a trivial task since there are two contradictory goals have to be satisfied simultaneously. A product line must not only be flexible in order to allow for diverse product instantiations, but also provide adequate generic components that can be used to create individual products with minimal effort [PERR98].

Two key issues in software product line development are *requirements analysis* and *variability realization* [WL99]. Domain analysis has been proven to be one efficient way to analyze and capture the requirements of a software product line [CN02]. Decisions about developing a product line within a domain can be based upon it.

Jacobson et al. [JGJ97] present a list of available techniques to implement variability, which are summarized as follows:

- *Inheritance* is used when the variation is a method that needs to be implemented for every application, or when some applications need to extend a type with additional functionality. Everything that is common to the new application is reused and others are replaced or extended through overriding.
- *Parameterization* is used when unbound parameters or macro expressions can be inserted into the code and instantiated later by binding the actual parameter or by expanding the macro. Template constructs in C++ can be used for parameterization. Pre-processor directives are another feature of C++, which enable more fine-grained configuration management.
- *Configuration* is used to select appropriate files and fill in some of the unbound parameters to connect components to each other. Source code is selected from code repositories and put together to form a particular product. The final configuration is usually performed by compile utility, such as make files.
- *Generation* of derived components is used when there is a higher-level language that can be used to specify a particular task. The language is then used to create the actual component.

2.2.4 Other Work

Lucent [WL99] introduced the FAST (Family-Oriented Abstraction, Specification, and Translation) methodology for product line engineering process. It divides domain engineering into domain analysis and domain implementation. Therefore, the issues involved in domain design are considered in the domain analysis phase. FAST promotes very small product lines, which are well understood, so development is a one-increment activity. The methodology provides the systematic guidance to each step during the product line engineering. These steps are carried out as transitions between process states, each of which is a group of activities that are performed in a particular situation to satisfy a specific concern.

Organization Domain Modeling (ODM) [STAR96] is a detailed domain analysis process with a set of work products and dossiers. ODM mainly concentrates on the

domain engineering of legacy systems although it is also capable of analyzing the requirements of new systems. It combines different artefacts such as requirements, design, and code from several legacy systems into reusable common assets. ODM is configurable with dossiers and can be integrated into other technologies. ODM is supported by DAGAR (Domain Architecture-based Generation for Ada Reuse) [KS96]. The DAGAR process applies ODM for domain modeling since it does not include that phase. DAGAR consists of activities for both domain engineering and application engineering.

Other approaches, FODA [KCH+90] (see Section 2.4.1), FeatuRSEB [GFA98] (see Section 2.4.3), FORM [KKL+98] (see Section 2.4.2), and RSEB [JGJ97] (see Section 2.8.2), will be introduced in the later sections.

2.3 UML Design Models

Software engineers are human beings. There are limits to the human ability to understand complexity. Modeling is a well-accepted engineering technique that narrows the problem which is being addressed [BRJ99]. In this section, we will at first introduce the basic concepts of UML. Part 2 discusses the UML structural models. Part 3 discusses the UML behavioural models. The last part discusses the UML extension mechanisms. We only focus on the UML design model and their notations, which are used in the methodology and the case study.

2.3.1 Unified Modeling Language

A *metamodel* is a precise definition of the constructs and rules needed for creating semantic models [OMG03]. A metamodel defines the language to specify a model. A *model* is a simplification of reality and provides the blueprints of a system [BRJ99]. Models help the modellers to concentrate the important aspects of a system in a less complexity form through discarding things, which are not concerned in those models. Software models are used to visualize, specify, construct, and document software systems. For different aspects of software systems, there are different types of models constructed to express details in different levels of abstraction [RJB99].

A *modeling language* is a language whose vocabulary and rules are used to convey the conceptual and physical aspects of a system [BRJ99]. The *Unified Modeling*

Language (UML) is a standard graphical language to model object-oriented systems. UML originates from the “unification” of Booch, Rumbaugh and Jacobson’s object-oriented modeling methods [RJB99]. It has rapidly become an industrial standard for software modeling [RATU03]. The current custodian of the UML is the Object Management Group (OMG) [OMG03].

Given an object-oriented system, the UML *structural models* identify the components and their relationships; the *behavioural models* capture the dynamic activities among the instances of those components. The *model extension* in UML enable modellers to add new kinds of modeling elements and attach free-form information to those elements [OMG03].

2.3.2 Structural Models

A *class* represents a concept within the system being modeled. A class has attributes, operations and relationships to other elements in the model. It is drawn as a solid-outline rectangle with three compartments separated by horizontal lines, to hold the name, attributes, and operations of the class. The signature of an operation may be italicized to indicate the operation is abstract. An *object* represents an instance of a class. An object has identity and attribute values. It is depicted similarly to a class, but with instance-like characteristics. The top compartment shows the name of the object and its class with underlined format, using the syntax: *object name: class name*. The second compartment shows the attributes for the object and their values, using the syntax: *attribute name: type = value*.

An *association* connects exactly two classes. It is depicted as a solid line connecting the two class symbols (or two different ends at one same class). A name string can be used to indicate the meaning of the path. If an association has class-like properties, such as attributes, operations, it can be modeled as an *association class*, which is represented as a class symbol which is attached to the association line with a dashed line. An association can be adorned by different kinds of optional property adornments, two of which are multiplicity and role names. A *multiplicity* property indicates the allowable range of the cardinality of the set of instances of the classes that the association connects. A multiplicity is represented as *lower bound .. upper bound*. A star (*) can be used to represent unlimited non-negative integer range. A *role* is represented by a name string near the end of the association path. It indicates

the role played by the class attached to the association end near the role name. A *link* is an instance of an association. The notation of a link is same as that of an association, but connects the instances of the two classes.

Inheritance in object-oriented paradigm is represented by a *generalization* relationship. It is shown as a solid-line path from the child to the parent, with a large hollow triangle at the end of the path pointing to the parent. An *aggregation* relationship is represented as a hollow diamond attached to the end of the association path pointing to the components. *Composition* aggregation is a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility to manage its parts. It is depicted by a solid filled diamond as an association end adornment. A *dependency* indicates a “semantic” relationship between two model elements: a change to the target element may require a change to the source element in the dependency. A dependency is shown as a dashed arrow between the two model elements, with an arrowhead pointing to the element on which the other element depends.

An *interface* specifies the externally visible behaviour of a class or a component without exposing the internal structure. One class or component may have multiple interfaces, each of which only specifies a limited part of the behaviour. Interfaces only have operations. An interface may be shown with a full rectangle symbol with two compartments and the keyword <<interface>>. A list of operations supported by the interface is placed in the operation compartment. Sometimes an interface is depicted by a small circle.

A *collaboration* describes how an element is realized by others in a specific way. The collaboration defines a set of roles to be played by instances and a set of interactions that define the communication between the instances when they play the roles. A collaboration is rendered as a dashed ellipse containing the name of the collaboration. A *package* is a grouping of model elements. A package may contain subordinate packages as well as other kinds of elements. All kinds of UML model elements can be organized into packages. A package is shown as a large rectangle with a small rectangle attached to the left side of the top of the large rectangle. A *note* is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. It is depicted as a rectangle with a dog-eared corner, together with a textual or graphical comment. A note is a common UML mechanism and can

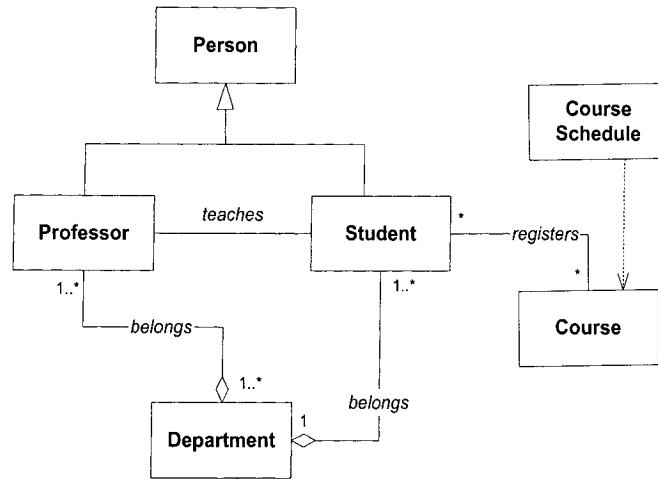


Figure 3: A Class Diagram Example

be used in both structural and behavioural models.

Figure 3 shows a simple class diagram. An instance of the **Department** class includes more than one instances of the **Professor** class or **Student** class. The **Person** class is the superclass of the **Professor** class and the **Student** class. They have a “teaches” association. A **Student** instance can register more than one **Course** instances, and one **Course** instance can be registered by more than one instances of the **Student** class. The **CourseSchedule** class depends on the **Course** class, i.e. changes made on the **Course** class may have impact on the **CourseSchedule** class.

2.3.3 Behaviour Models

A *sequence* diagram presents a set of messages between instances to perform a desired service. There are two dimensions in a sequence diagram: the vertical dimension represents time, and the horizontal dimension represents different instances. Typically, time proceeds from top to bottom, and there is no significance to the horizontal ordering of the instances. In a sequence diagram, an object box with its vertical dashed line represents an object. A *focus of control* shows the period of time during which an object is performing an action. It is depicted as a tall, thin rectangle.

A *collaboration* diagram presents the collaboration amongst a set of instances and their relationships given in a particular context. The notation of a collaboration diagram is similar to the notation of an object diagram. The difference between them is that, the links in a collaboration diagram are adorned with messages the objects

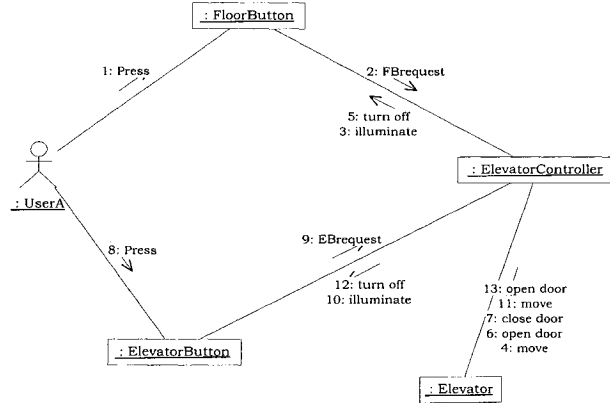


Figure 4: A Collaboration Diagram Example

send and receive, and the order of the interaction is described with a sequence of numbers starting with the number 1.

Figure 4 shows a collaboration diagram of Elevator System. When a User presses a FloorButton, a Fbrequest message is sent to an ElevatorController object. The ElevatorController illuminates the pressed FloorButton and moves the Elevator towards the User. The FloorButton is turned off once the Elevator arrives at the floor at which the User is. The door of the Elevator opens to let the User steps in. The ElevatorController closes the door after a period time. The User presses the ElevatorButton to specify the destination floor. The EBrequest is sent to the ElevatorController by the ElevatorButton. It is then illuminated by the message sent from the ElevatorController. The ElevatorController moves the Elevator to the destination floor. Once the Elevator arrives, the ElevatorButton is turned off and the door is opened to let the User step out.

A *statechart* diagram is used to describe the behaviour of an object or an interaction. It focuses on the possible sequences of states and actions through which the object or the interaction can proceed during its lifetime because of reacting to discrete events. A statechart diagram is a graphic notation for a finite state machine. A *state* is represented as a rectangle with rounded corners. An *initial state*, which indicates the default starting place of the state machine, is depicted as a filled black circle. A *final state*, which indicates the execution of the state machine, is represented as a filled black circle surrounded by an unfilled circle. Note, initial and final states are pseudo-states and may not have usual parts of a normal state except for a name. A

transition is a relationship between the two states indicating that the object described by the state machine will traverse from the source state to the target state when a specified event occurs and specified conditions are satisfied. A transition is rendered as a solid directed line from the source state to the target state.

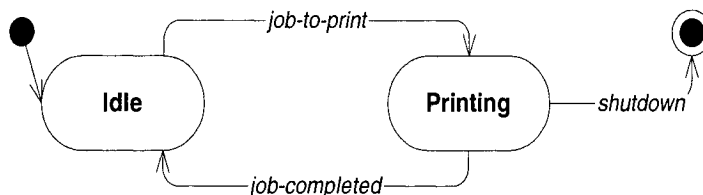


Figure 5: A Statechart Diagram Example

Figure 5 gives a simple example of statechart diagrams. It describes the states of a printer. When a *job-to-print* event is triggered, the printer shifts its state from *Idle* to *Printing*, and recovers to the *Idle* state once the printing job is completed.

An *activity* diagram is a special case of a state diagram in which all or most of the states are action states and in which all or most of the transitions are triggered by completion of the actions. Activity diagrams focus on flows driven by internal processing rather than external events. The notations used in an activity diagram are almost as same as those of a state diagram, except the forking representation of *branches*.

Figure 6 shows an activity diagram example. It describes the state transition of an Order Processing system. The system displays the login screen for the users to enter name and password for authentication. Users can perform various activities regarding with orders: Place order, Display order status, and Cancel order. Users choose "exit" to terminate the system.

2.3.4 Model Extension

Stereotypes, tagged values, and constraints are the extension mechanisms of UML [OMG03]. A *stereotype* is an extension of the vocabulary of UML, allowing modellers to create new kinds of modeling elements similar to existing ones. A stereotype is rendered as a name enclosed by guillemets, and placed above the name of another element. A *tagged value* is an extension of the properties of an UML element, allowing modellers to create new information in the specification of the element. A tagged value

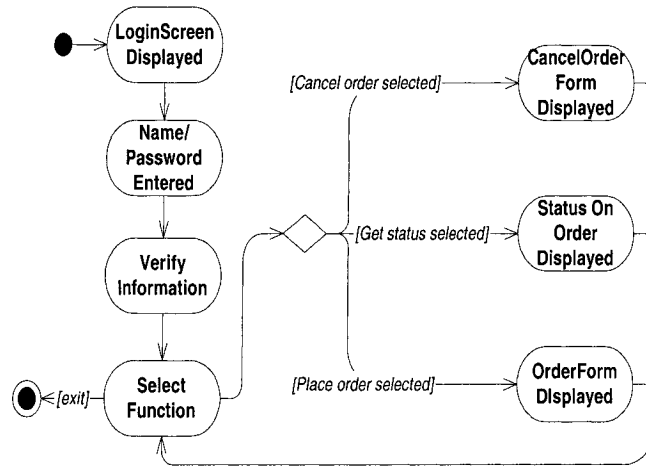


Figure 6: An Activity Diagram Example

is depicted as a string enclosed by brackets and placed below the name of another element. A *constraint* is an extension of the semantics of an UML element, allowing modellers to add new rules or to modify existing ones. Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to the elements by dependency relationships. A constraint can be also rendered as a note.

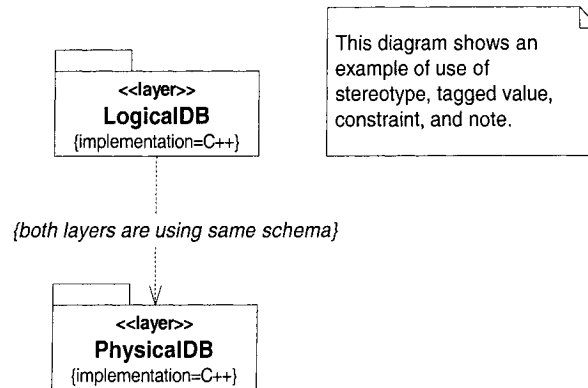


Figure 7: An Example of UML Extensibility

In Figure 7, a new modeling element, Layer is defined as a stereotype, based on a package. The LogicalDB layer depends upon the PhysicalDB layer. Both of them have a tagged value, which indicates that the layers are implemented with C++ language. A constraint in natural language bound to the dependency between the two layers. The note explains the purpose of the diagram.

2.4 Feature Model

As one of the widely recognized domain analysis techniques, *feature modeling* is “the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a feature model” [KCH+90]. A *feature model* provides an overview of requirements, distinguishes between common and variable properties, shows dependencies between features, and enables feature selection to define new products. It also helps developers define reusable components and describe dependencies between components and features. Feature modeling is particularly important for reusable software development because reusable software contains inherently more variability than typical non-reusable ones. Feature models have already been applied in large industrial projects in many domains, such as Electronic Bulletin Board [BRYA94], Telecommunication [VAM+98] [KKLL99], and Elevator systems.

In this section, we will at first introduce FODA since it is the cornerstone of feature model research. Part 2 discusses another famous method, FORM. Other work in the literature is given in the last part.

2.4.1 Feature Oriented Domain Analysis

Feature Oriented Domain Analysis (FODA) [KCH+90] was developed at Software Engineering Institute. Its thorough description of the domain analysis process and explicit feature modeling has become the foundation for subsequent work in the research area [KKL+98] [GFA98] [KLL+02] [PR04].

In FODA, a *feature* is “a prominent or distinctive user visible aspect, quality, or characteristic of a software system or systems” [KCH+90]. It represents an important property of a concept in a domain. A *concept* can be anything of interest in a domain. Features can be identified from existing and potential customers, domain experts and literature, and exemplar applications. Feature analysis covers a broader range than traditional requirement analysis does, since features cover both functional and non-functional requirements, even non-technical constraints such as business laws in the domain.

There are three kinds of features: mandatory features, optional features, and alternative features. Given a domain, a *mandatory feature* is the core property of a

concept, a main domain characteristic, and constitutes the domain infrastructure. An *optional feature* represents a property that may not be necessary to some applications in the domain. Optional features indicate secondary properties of the domain in contrast with the primary properties represented by mandatory features. *Alternative features* represent different ways to configure a mandatory or optional feature. Both optional features and alternative features are *variable features*.

A feature can be an *aggregation* (super-feature) of other features (sub-feature). A *feature set* is composed of features, and the *constraints* over the features: if the super-feature of a mandatory feature is included in a feature set, the mandatory feature must be included in the feature set; if the super-feature of an optional feature is included in a feature set, the optional feature may or may not be included in the feature set; only one alternative feature is included in the feature set to which the super-feature of the alternative feature belongs. A feature is a *variation point* if it has at least one variable sub-feature. *Composition rules* specify the dependencies between variable features in a feature set: two features must be selected together if there is a “*requires*” dependency between them; if the selection of two features are mutual exclusive, there is a “*mutex-with*” dependency between the two of them.

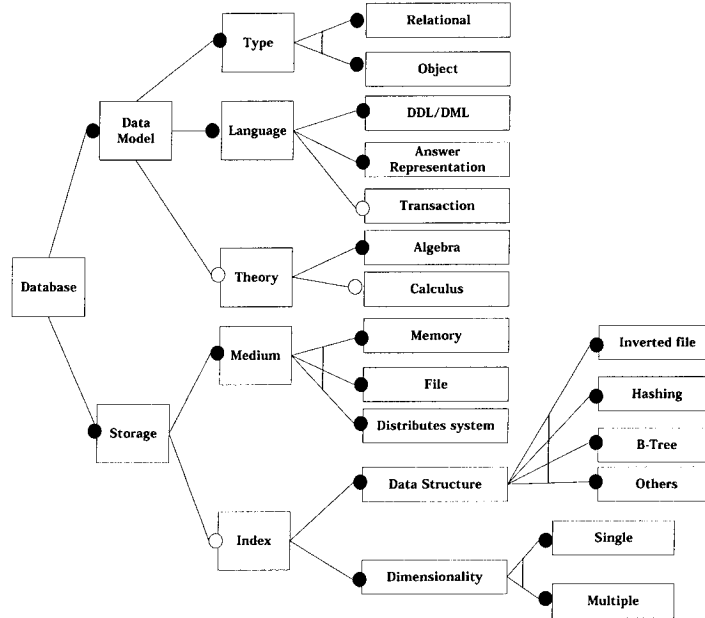


Figure 8: A Feature Diagram of the Database Domain

A *feature diagram* is the graphic notation of a feature set. A feature is depicted with a rectangle. A mandatory feature is depicted by attaching a filled circle to the rectangle. An optional feature is depicted by attaching an empty circle to the rectangle. An aggregation is illustrated by lines from the super-features to all the sub-features. Alternative features are portrayed with an arc across the aggregation relationship. The root of a feature diagram, called “concept node”, represents the domain concept being modeled. All features in the diagram except the concept node are properties of the concept. For a feature diagram of a product line, the position of a feature within the hierarchy shows its influence on the product line architecture. The composition rules are expressed as: “{feature1} (‘requires’ | ‘mutex-with’) {feature2}”.

A feature diagram is usually accompanied by additional information, such as semantic description of features, feature selection rationale, available exemplar systems, etc [KCH+90]. A feature model is constituted by feature diagrams and the additional information. Typically, a feature model focuses on identifying properties, factors, and assumptions that can characterize products of a product line or differentiate one product from others in the same product line, rather than finding implementation detail of the products.

Figure 8 shows a simplified feature diagram of the database domain. The concept node is the **Database** feature. Properties such as “**Language**”, “**Storage**” are mandatory features, and they appear in any instance of the concept, i.e. applications in the domain. On the other hand, the “**Index**”, “**Transaction**” are optional features, and may not exist in some applications. The “**Data Structure**” feature has a variation point, which can be configured to B-tree, hashing, or other mechanisms.

2.4.2 Feature Oriented Reuse Method

The Feature Oriented Reuse Method (FORM) [KKL+98] extends FODA to the design phase and illustrates how the feature model is used to develop the domain architecture and components for reuse, under a set of guidelines. Features are organized into four categories in order to facilitate mappings from the feature model to design artefacts:

- *Capability* features literally characterize services, operations, and non-functional constraints of applications in a given domain. For instance, query processing is a capability feature of a DBMS application.

- *Operating Environment* features represent the environmental constraints from hardware and software aspect. For example, a DBMS application may need 100GB hard disk spaces.
- *Domain Technology* features are a set of concepts, terminology, domain specific methods and standards, laws, which are used for communication in a given domain. For example, a B-tree index structure in the DBMS domain.
- *Implementation Technique* features represent low-level implementation issues, such as an Abstract Data Structure (ADT) or a communication protocol. These features are more “general” compared with the domain technology features.

Feature identification in FORM is also classified accordingly [KKL+98]. Typically, capability features are mainly identified from user manuals; the requirement and design documents are good for finding domain technology and operating environment features; implementation features can be found in the design documents and source codes.

There are three types of feature relationships in FORM. The *composed-of* relationship is same as the aggregation relationship in FODA. A *generalization* relationship indicates that a child feature can appear any place its parent feature can. When a feature is the prerequisite to implement another feature, there is an *implemented-by* relationship between them.

FORM consists of two engineering processes: domain engineering and application engineering [KKL+98]. The *domain engineering* process starts with domain analysis to find the commonality and variability across the applications in a given domain. The analysis results in a feature model, which is used to develop reference architectures and reusable components of the domain. A reference architecture is a common software architecture for a family of applications in a domain [TRAC94]. The architecture of an individual application in the family can be obtained by instantiating the reference architecture. However, a reference architecture is different from a Domain Specific Software Architecture (DSSA) [TRAC94]. Typically, a DSSA concentrates on the process to develop and instantiate reference architectures, whereas a reference architecture focuses on the structure itself.

In FORM, a reference architecture is defined from three viewpoints: subsystem, process, and module. A *subsystem* architecture groups service features and allocates

them to different hardware. Each subsystem is further decomposed into *processes* which are concerned with the operating environment features. *Modules* are derived from the domain technology and implementation technique features. The *application engineering* process develops applications with the artefacts created in the domain engineering. Typically, the efforts of building an application can be leveraged by the customization of reference architectures with feature selection in a mature and stable domain [KKL+98].

2.4.3 Other Work

Griss et al. [GFA98] integrates the standard FODA process into the RSEB methodology [JGJ97] (see Section 2.8.2) to form the FeatuRSEB methodology. Instead of the use case models in RSEB, FeatuRSEB uses feature models to play the unifying role of models. These feature models act as a convenient centre repository to store features for re-users to develop applications. FeatuRSEB also proposes an approach to implement the feature diagram notation with the predefined UML modeling elements. The feature models in FeatuRSEB extend those in the original FODA by distinguishing between OR and XOR alternatives, where XOR shows mutual exclusion and OR enables more than one feature. Czarnecki [CE00] also integrates the “OR” feature into the feature models for generative programming. He defines a set of normalization rules to normalize feature diagrams.

Kuusela [KS00] divides software requirements into design objectives and design decisions. *Design objective* features are properties related to the functional requirements. They are presented within a design objective feature model, which is quite similar to the one described by FODA. The features are either mandatory or optional, presented in a tree structure with “decomposed-to”, “requires”, and “excludes” relationships. A concrete application is represented by a subset of those features. The *design decisions* reflect solutions of the requirements and capture the design rationale. The nodes in the design decision feature model represent solutions, such as data structures, design patterns, third party components or architectural decisions.

Riebisch [RBSP02] argues that the XOR and OR notation bring ambiguities into feature models, and are not expressive enough in some circumstances. He introduces the concept of “multiplicities”, which are similar to those of UML, to group features. For example, “0..1” means at most one feature can be chosen from the set of the

sub-features.

Philippow et al. [PR01] propose a way to maintain traceability between feature models and design models with feature names. A new stereotype $\langle\langle\text{variant}\rangle\rangle$ is introduced to model variable features. Each element of the new type is annotated with a tag value which has a key “feature”. The key’s value is the feature name. They claim that variant elements in design models refer to the features in the feature model with the names, to achieve the traceability.

The feature concept has been integrated into UML since version 1.4 [OMG01]. However, a feature in UML is a property similar to an operation or an attribute, and is encapsulated in an interface or a class. It is different to the feature concept in feature models.

2.5 Use Case Model

A *use case* “specifies the behaviour of a system or a part of a system and is a description of a set of sequences of actions, including variants that a system performs to yield an observable result of value to an actor” [JBR99]. An *actor* defines “a coherent set of roles that users of an entity can play when interacting with the entity.” [OMG03] Actors may be users of the software being modeled, or the operating environment, with which the software must interact, or even part of the software [BRJ99]. A use case can be viewed as a description of one specific use of the software by an actor. It includes a set of sequences, in which each sequence represents the interaction between actors and the software itself. Use cases are defined from the users’ point of views without exposing unnecessary design or implementation details. A use case can be specified in natural languages and provides a good way to clarify software requirements between the developers and the clients [PREE92].

In this section, we will at first introduce the UML use case model and the notation. Part 2 discusses other work and concepts in the research area.

2.5.1 Use Case in UML

UML integrates use case modeling and provides it a set of graphic notations, which are widely accepted in the software community [JBR99] [RATU03]. A use case is depicted as an ellipse, within which a unique text string is put inside as the name of

the use case. An actor is depicted as a stick man figure with the name of the actor below the figure. Use cases and actors are connected with *associations* to express the interaction between them. An association is symbolized as a straight line. UML defines three relationships among use cases or actors:

- *Include*: If one use case incorporates the behaviour of another use case to fulfill its behaviour, they have an include relationship. The former use case is called a “base use case”, and the latter is called an “included use case”. The location of the included behaviour is specified in the base use case. An include relationship is depicted as a dashed line with an open arrowhead from the base use case to the included use case. The arrow is labelled with the keyword <<include>>.
- *Extend*: An extend relationship from use case A to use case B indicates that the behaviour of B may be augmented (subject to specific conditions in the extension) by the behaviour specified by A. The behaviour is inserted at the *extension point* in B. An extend relationship is shown by a dashed arrow with an open arrowhead from the extending use case to the base use case. The arrow is labelled with the keyword <<extend>>. The condition of the relationship may be presented close to the keyword.
- *Generalization*: A *child* use case inherits all the attributes, sequences of behaviour, and extension points defined in its *parent* use case. A child use case can also override the behaviour of its parent, add new behaviour, and participate all relationships of the parent use case. A generalization relationship is depicted as a solid directed line with a large open arrowhead pointing to the parent use case. The generalization relationship between actors has the same semantics and notation.

A *use case diagram* illustrates a set of use cases and actors and their relationships [BRJ99]. In Figure 9, a User is the actor of the DBMS Application. The Process SQL query use case and Process OQL query use case are the children use case of the Process Query use case. The Process Query use case includes the behaviour of the Optimize Query use case. The Bottom-Up Query Optimization use case and Transformative Query Optimization use case extend the Optimize Query use case at the extension point strategy.

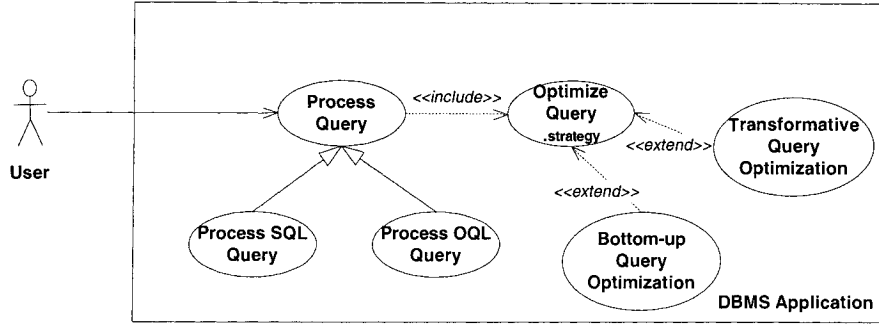


Figure 9: Use Case Example

A use case describes a set of sequences, each of which represents one possible way to carry out the behaviour described by the use case. Each sequence is called a *scenario* [BRJ99]. A use case encompasses a collection of scenarios. A scenario can be described in several ways, from a simple narrative text description, to numbered steps indicating the “*subject, action, object*” triples, or the UML sequence or collaboration diagrams. A scenario is basically one instance of a use case. UML divides scenarios into: *primary scenario*, the essential sequences; and *secondary scenario*, alternative sequences [BRJ99].

2.5.2 Other Work

Regarded as the inventor of use case, Jacobson worked with colleagues [JCJO92] to provide a thorough methodology addressing architectural, process, and organizational aspects of software reuse. The requirements are gathered in a way from informal scenarios to the refined use case models. Each scenario is viewed as a test case of the target system. Variability is captured with variation points using three mechanisms: *inheritance* is used to specialize or extend behaviour with the $\langle\langle\text{uses}\rangle\rangle$ and $\langle\langle\text{extends}\rangle\rangle$ stereotypes; *configuration* slots are filled by choosing alternative component implementations; *parameterization* takes the form of a bound variable, a template instantiation, or an evaluated expression.

Potts et al. [PTA94] view a scenario as a way an actor attempt a task that is specified by a use case. They define different kinds of scenarios. A *main scenario* describes the usual way in which the task is successfully performed. Typically, in a main scenario, the simplest sequence of interactions to execute the task is described, and all

steps of the sequence are assumed execute successfully. A *variant scenario* describes another way to perform the task and all steps are assumed execute successfully. An *exceptional scenario* describes a scenario where exceptional or error conditions may arise during the task execution. It is possible to recover from the exceptions and therefore successfully complete the task, which is described in a *recovery scenario*; if not, a *failure scenario* is used for the description. All kinds of scenarios other than main scenarios are *secondary scenarios*. The analysis of a software system is based upon the thorough understanding of the scenarios derived from use cases.

A use case can also be viewed as a description of a cohesive set of *dialogues* that the actor initiates with the system [BCKR97]. The dialogues are *cohesive* in the sense that they are related to the same task, or form part of the same transaction. Cohesiveness is often determined by having a *goal* in common for the tasks, or by having a common *responsibility*.

Cockburn [COCK97] identifies four dimensions in use case descriptions: *purpose*, *content*, *plurality*, and *structure*. A *purpose* dimension can be either a user story or a requirement. A *content* dimension can be either informal contradicting prose or formal contents. A *plurality* dimension indicates the multiplicity of scenarios, while a *structure* dimension can be unstructured, semi-formal, or formal structure. He [COCK97] claims his own approach is “requirements, consistent prose, multiple scenario, and semi formal structure”.

Use Case Map (UCM) [BUHR98] is a visual notation for comprehending and developing the architecture for emergent behaviour in large, complex, self-adapting systems. A UCM is a two dimensional map of cause-effect chains from points of stimuli through the system to points where responses are observed.

The OPEN Modeling Language (OML) [FHG98] is a competing meta-modeling language to UML. It represents the merger of SOMA [HFG97], MOSES [FHG98], and Firesmith [HF97]. One important aspect of OML is the notion of *tasks* and techniques. In OML, a task-action grammar is defined, so the requirement identification involves task scripts.

Andersson and Bergstrand [AB97] present a method to formalize use cases to have unambiguous syntax with Message Sequence Charts (MSC). This method divides use case models into three levels. The *system* level describes the functional view of the system; the *structure* level describes the use case behaviour without going into detail

by utilizing a hierarchical view that allows information hiding; and the *basic* level describes the detailed interaction between the system and actors. The UML sequence diagrams are used as notations.

Regnell [REGN99] focuses on the role of use case modeling in requirement engineering and its relation to system verification and validation. He defines the process of Usage-Oriented Requirements Engineering, an extension of use case driven analysis; and Synthesized Usage Model, the output of the process. He also proposes a use case metamodel, which characterizes the three levels in a use case. The *environment* level identifies the relationship between use cases with external entities. The *structure* level describes the internal structure of use cases. A use case is described in terms of *episodes* [RAB96]. Each episode represents a subtask. An episode is composed of *events*, each of which is a significant occurrence that has a location in time and space. The *event* level distinguishes the specialized types of events.

2.6 Software Architectural Model

Software architecture is the high level structure of a software system. It defines “a system in terms of computational components and interactions among those components” [SG96]. In this section, we focus on the description of software architecture with multi-view architectural models. The first part introduces the concepts of architecture styles and architectural models. Part 2 discusses the “4+1 view” approach since it is the cornerstone of this research area. The last part introduces the applied software architecture.

2.6.1 Architecture Model

Shaw and Garlan [SG96] propose the concept of *architectural style* to define common software architecture. An architectural style is described in terms of components, the description of the elements from which systems are built; connectors, the interactions among those elements; configuration rules, the constraints of how components and connectors may be configured; semantic interpretation, which defines when suitably configured designs have a well-defined meaning as architecture; and analyses, which may be performed on well-defined designs. Widely used architectural styles include pipe and filter, client-server, object-oriented, and layered architecture [SG96].

Architecture designers can choose a style based upon the system requirements.

An *architectural model* is the high-level design abstraction of a software system in graphical documentation [KRUC95]. Architectural models document architecture to enable communication on the architecture among stakeholders, to capture early design decisions, and to provide reusable abstractions of software systems [BCK97]. Kruchten [KRUC95] suggests use multiple, concurrent diagrams to describe the entire software architecture of a system, in order to overcome problems such as crowded diagrams, inconsistent notation, and missed requirements, etc. Classification of the diagrams are based on the perspectives, also called *views*, of different groups of stakeholders of the system. An architecture view specifies the needs on the architecture from a specific group.

Among the fundamental work on multi-view architectural modeling, the “4+1 view” approach proposed by Kruchten [KRUC95], and the *applied software architecture* approach proposed by Hofmeister et al. [HNS99], have attracted interest from industry and academia [MT00].

2.6.2 “4+1” View

The “4+1” view approach organizes a description of software architecture with five concurrent views, each of which addresses a specific set of concerns. The *logical view* concerns the solutions to functional requirements. The *process view* focuses on dynamic aspects of the model and describes runtime behaviour, such as thread of control. The *physical view* describes the mapping of the software onto the hardware and reflects its distributed aspect. It considers the system’s non-functional requirements. The *development view* focuses on the actual module organization and the software development environment. It is concerned with the ease of development, software management, reuse or commonality, and to the constraints imposed by the toolset or the programming language. The “+1” stands for *scenarios*, which are used to unify the elements of the four views. Requirements are specified with scenarios. Scenarios help designers discover architectural elements during the architecture design, and validate the design [KRUC95].

Architecture design with the “4+1” view is an evolving process [KRUC95]. At the beginning, scenarios are chosen based on risk and criticality, and a coarse architecture of basic elements is created. The elements are organized into the four views. In the

next iteration, risks are reassessed and scenarios may be extended. The preliminary architecture is reviewed and probably additional elements or significant architectural changes are made. The four views are updated accordingly. In the same time, opportunities of reuse and identification of commonality are also considered. New iteration starts and the process evolve until the architecture is stable. These views are carried over into the UML system modeling [BRJ99].

Kruchten [KRUC95] mentions that not all software architecture need the full set of views. A view can be omitted from the architectural model if the view is insignificant. For example, it is not necessary to have the physical view if there is only one processor; the process view is useless for a single process system; and the logical view can be combined with the development view for very small systems.

2.6.3 Applied Software Architecture

Applied software architecture, also called the Siemens approach, is a result of a study into the industrial practices of software architecture [HNS99]. It consists of four views modeled in UML notation, as shown in Figure 10 [HNS99]. The *conceptual view* presents the configuration of components and connectors. The *module view* shows the structure in terms of layers, subsystems, modules and their interfaces. The *execution view* identifies the hardware resources, communication mechanisms and paths, and the runtime entities such as processes. The *code view* presents the organization of source code, libraries, binaries, and executables.

The four views are loosely coupled. The components and connectors identified in the conceptual views are used as the rationale in the module view to design the modules, subsystems, and layers. The artefacts produced in the module view are consulted in the execution view, and the artefacts in the execution view are consulted in the code view.

Among the four views, the module view is close to the traditional architectural description in terms of layers and subsystems. The main purpose of the module view is to map the functions and responsibilities to different modules. A *module* encapsulates data and operations to provide services through its own interface. Modules can be grouped into a subsystem, or assigned to a layer. A module can contain other modules. A *subsystem* is a group of high coupling modules. Typically, modules grouped into a containing module are more tightly coupled than the modules contained in

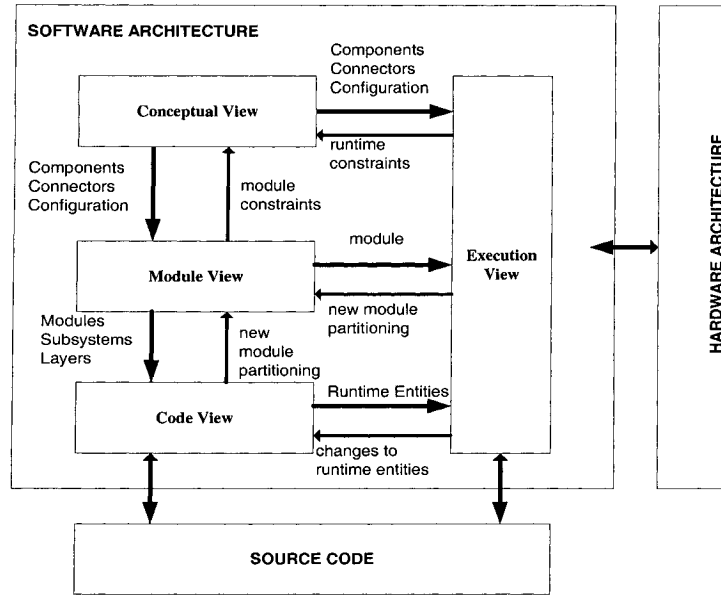


Figure 10: The Four Views of Applied Software Architecture

a subsystem. *Layers* organize modules into a partially ordered hierarchy. A layer can contain other layers. An *interface* is a collection of operations. Interfaces act as connection points for layers and modules. Subsystems are not allowed to connect with interfaces. There are two kinds of interfaces: “*provide*” interface, and “*require*” interface. A module can only directly use other modules in the same layer. In order to use modules in other layers, the module has to use the interface of the layer, to which those required modules belong. The metamodel of the module view is shown in Figure 11 [HNS99].

2.7 Design Pattern

Building software is similar to building a house. Bricks, woods, and cements are composed together in different ways to construct houses. Common ways can be found in the construction of a series of buildings of similar purposes. These ways are referred as “patterns”. A *design pattern* describes a commonly recurring structure of communicating components that solves a general design problem within a particular

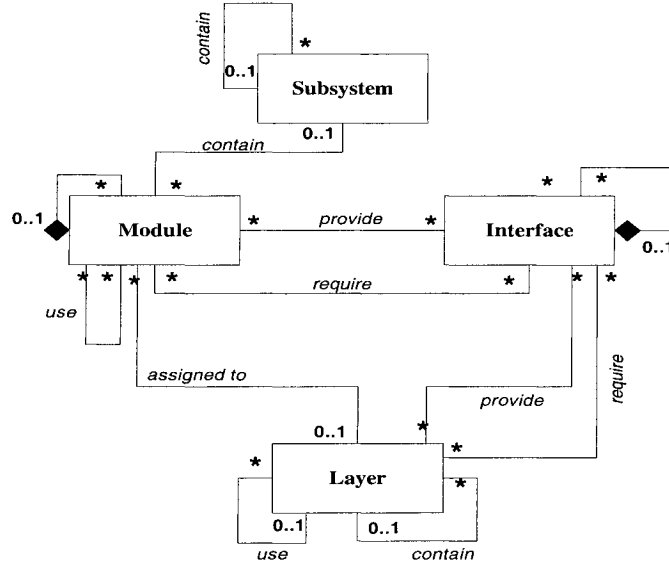


Figure 11: The Module View Metamodel

context [GOF94]. In this section, we will at first introduces the basic concepts of design patterns. Then we give an overview on object oriented design patterns since they are made heavy use in framework design and documentation [SCHM97] [JOHN92].

2.7.1 Pattern

Alexander [AIS77] defines the concept of *pattern* as the description of a recurring problem, and the reusable solution to that problem. Four elements are identified for a pattern: a unique *name* for communication, the *problem* described with pre- and post-conditions, the abstraction of the *solution*, and the *consequence* of applying the pattern.

Buschmann [BMR+96] categorizes software patterns into three groups in terms of their level of abstraction. An *architectural pattern* provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines to organize their relationships. A *design pattern* describes a commonly recurring structure of communicating components that solves a general design problem within a particular context. Architectural and design patterns are independent of the implementation languages. An *idiom* is a low-level pattern specific to a programming language. It describes the way to implement particular aspects of components or the relationships

between them using the features of the given language. Software patterns capture design expertise to shorten the path from novel programmer to experienced designer, facilitate reuse across applications, and provide common vocabulary for the communication of designers.

2.7.2 Object Oriented Design Pattern

Due to the overwhelming acceptance of the “Gang of Four” book [GOF94], much of the patterns focused by the software community are object oriented design patterns. An *object-oriented design pattern* names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. It describes the roles and responsibilities of the objects and classes that collaborate to solve a general design problem in a particular context. The description also includes the applicable scenarios, the constraints, and the consequences of applying the design pattern. Design patterns are classified into three categories: *creational patterns* are concerned with initializing and configuring classes and objects; *structural patterns* are capable of decoupling interface from implementation of classes and objects; and *behavioural patterns* handle dynamic interactions among classes and objects. It should be noted that we refer to an object-oriented design pattern as “design pattern” throughout the dissertation.

It is difficult to understand design patterns with many active objects. To solve this problem, Riehle and Gross [RG98] propose role models to describe the dynamic aspects of design patterns. A *role* represents the view an object holds on other objects and focuses on only one particular aspect of the object. An object may play several roles at once, and the same role may be played by different objects. A *role model* describes a particular aspect of object collaboration in a design pattern as a set of roles and their relationships. Objects achieve the purpose of a role model by acting as the definition of the roles they play.

Design patterns are frequently used in software product lines and frameworks since they provide an effective way to realize variability [JOHN92] [BJ94] [FSJ99]. Roberts and Johnson [RJ97] suggest a list of design patterns and the corresponding variability they address (Table 1). For example, if a framework has to provide the flexibility of allowing the application developers to decide specific traversal algorithms for an aggregate object, the *Iterator* pattern can be chosen. The pattern takes the

Variability	Design Patterns
Algorithms	Strategy, Visitor
Actions	Command
Implementations	Bridge
Response to change	Observer
Interaction between objects	Mediator
Object creation	Factory, Prototype
Structure creation	Builder
Traversal algorithm	Iterator
Object interfaces	Adapter
Object behavior	Decorator, State

Table 1: Design Patterns for Variability

responsibility for access and traversal out of the aggregate object and put it into an iterator object. The common interface for the traversal can be defined into an abstract iterator class, and put it into the framework. Different traversal algorithms on an aggregate object can be realized by providing the subclasses to the iterator abstract class.

2.8 Framework

Object oriented application frameworks are believed to be the core of cutting-edge technology of the twenty-first century [FSJ99]. In this section, we will at first give an overview of framework and the related concepts. Part 2 introduces the existing framework development methodologies. Framework evolution is discussed in part 3, and framework documentation is described in the last part.

2.8.1 Framework Introduction

An *application framework* provides a generic design within a given domain and a reusable implementation of the design [JF88]. An *object-oriented application framework* presents its generic design and reusable implementation through a set of abstract classes and their collaborations [BJ94]. The design of the framework fixes certain roles and responsibilities amongst the classes, as well as standard protocols for their collaboration. Customizing a framework by subclassing the given abstract classes makes the

development of individual application cost-effectively. Frameworks are extensible and flexible so that new components can be built and easily fitted into the infrastructure. Typically, a framework is developed by expert designers who have deep knowledge of the application domain and long experiences of software design. Frameworks offer a concrete realization of a software product line [CN02].

The first widely used framework was the Smalltalk-80 user interface framework, called the Model/View/Controller (MVC) [KRAS88], which was developed in late 1970's. MVC divides the user interface into three parts; *models*, an application object that is independent of the user interface; *views*, which manages a region of the display; and *controllers*, which converts user input events into operations on its model and view. MVC was followed by other GUI frameworks such as ET++ from the University of Zurich [WGM88]. There are a number of large commercial frameworks such as Microsoft Foundation Class (MFC) [PROS99], Taligent [CP95], Java Abstract Window Toolkit (AWT) and its successor Swing [DFK04]. Frameworks can be built on other domains, such as Choices for operating systems [RUSS91], and MET++ for multimedia applications [ACKE96]. Frameworks do not need to restrict the implementation to object-oriented languages. The Genesis database system compiler is a framework for database management systems [BBG+89]. It is implemented in the C language.

Cotter and Potel [CP95] view a framework as three parts. The *framework core* is composed of abstract classes that define the generic structure and behaviour of the framework. The *framework library* comprises concrete components that can be used with little modification by applications developed from the framework. The *unused library classes* are optional for some applications built from the framework. The core and library are also called *ensemble*.

Variability between applications in a specific domain is identified in terms of hot spots. A *hot spot* is a variable aspect of an application domain with associated responsibilities [PG94]. A framework provides simple mechanisms to customize each hot spot that resides in the framework architecture to instantiate concrete applications. A hot spot may have many hooks within it. A *hook* is a place in a framework that can be adapted or extended in some way, such as by filling in parameters or creating subclasses, to provide application specific functionality [PREE94]. Hot spots are usually realized with design patterns [GOF94] [PREE99]. In contrast, *frozen spots*

represent commonality across applications.

Frameworks use template methods and hook methods to realize commonality and variability in source code [PG94]. A *template method* provides the generic algorithm or steps to perform a task. It calls one or more hook methods. A *hook method* represents a point of variability by providing the calling interface to a variable task. Each implementation of a hook method provides an alternative of that task. A template method defines a generic control flow that is composed of hook methods. An *abstract* hook method does not have the implementation of the method it defines, while a *regular* hook method does. A *template class* is a class that has a template method, and a *hook class* is a class that has a hook method [JR91]. A class can be both a template class and a hook class depending on different contexts. It should be noted that template methods in hot spots are different to the C++ template constructs [STRO97]. In the remainder of the thesis, we always use the term to refer to a template method in a hot spot if there is no additional description.

Design patterns can be considered as reusable micro-architectures that constitute a framework [GOF94]. They describe common and frequently observed relationships among classes, help determine the detail structure of hot spot subsystems, and compose the rationale of the framework design [JOHN92]. Design patterns present proven solutions for how to internally structure hot spot subsystems in detail [SCHM97]. Typically, design patterns are smaller architectures than frameworks. Frameworks use a high density of design pattern to provide flexibility and extensibility. Moreover, frameworks are always related to a specific application domain, and are more specialized than design patterns, which can be applied in any application domain [GOF94].

Framework reuse is different with class library reuse. A *class library* is a set of related classes that provides general purpose functionality [STRO97]. The classes in a class library are often not related to a specific application domain, which is the case for classes in a framework [JF88]. A class in a class library is reused individually, whereas a class in a framework is usually reused with other classes in the framework together under a prescribed protocol. Moreover, a framework is reused in the architectural level, while a class library is reused in the class level. Cotter and Potel [CP95] observe the following limitations in class library reuse:

1. The class hierarchies in large systems may be too complex to be comprehended by designers cost-effectively.

2. Parallel development with class libraries may result in different solutions to the same kinds of problems to obstruct maintenance activities.
3. The collaboration of objects created from class libraries in application development has to be decided by the developers and errors can be made in the process.

Johnson and Foote [JF88] define two types of frameworks from the customization perspective: *White-Box* framework, and *Black-Box* framework. White-Box framework customization relies on inheritance of existing classes and requires in-depth framework knowledge, while customization of Black-Box frameworks is performed via using composition of existing components, each of which understands a particular protocol. They claim that White-Box frameworks are hard to learn and require much more efforts to use, in contrast to Black-Box frameworks. A framework “becomes more reusable as the relationship between its parts is defined in terms of a protocol, instead of using inheritance” [JF88].

2.8.2 Framework Development

Developing a framework is different from developing an individual application because a framework has to cover all relevant concepts in a domain, while an application is only concerned with the application requirements [BMMB00]. Thus, standard software development methodologies are not sufficient for developing object-oriented frameworks [PG94]. Although there have been several methodologies suggested for the development of frameworks, the methodologies vary quite widely, and have been poorly supported by notations for models [FSJ99]. The existing methodologies can be classified into Bottom-Up, Top-Down, Hot Spot Generalization, and Use Case Driven.

2.8.2.1 Bottom-Up

An intuitive approach to design a framework is to begin with a White-Box framework, which is the result of generalizing from a number of concrete applications [JF88]. Once the first version of the framework is done, it will be easier to develop more examples [RJ97]. Wilson et al. [WW93] formalize the idea as:

1. Develop several applications that are planned to be built from the framework in the problem domain. A general rule is to build the first application; then build a second application that is slightly different from the first one; and finally build a third application that is even more different than the first two. This is often referred to as the “rule of three” [RJ97].
2. Identify the common features in the applications and extract them to construct a framework.
3. Redevelop those applications from the framework to verify the extracted features.
4. Evolve the framework and build more applications from the framework.
5. Iterate step 4 until the framework can handle all applications in the domain.

Because applications are built prior to developing a framework, the approach is often referred to as “bottom-up”. Johnson [JOHN93] states that bottom-up approaches emphasizes an incremental way to build a framework with iterative refactorings to restructure the framework (see Section 2.9.1). Nevertheless, he also recognizes that generalizing applications to abstract design are difficult and expensive.

2.8.2.2 Top-Down

Top-Down approaches are domain engineering methods. They start with domain analysis to organize commonality and variability within a domain into an analysis model [CHW98] [CE00] [CN02]. The model is used to define the Domain Specific System Architecture (DSSA) and appropriate reusable components that can be instantiated during actual application development. The DSSA can be instantiated to frameworks since frameworks are a kind of DSSA [TRAC94]. A framework is validated by building test applications from the framework and is revised upon the testing results. The development is an iterative process. The top-down approaches include ODM [STAR96], FORM [KKL+98], and FAST [WL99](see Section 2.2.4).

Valerio et al. [VSF97] propose a domain analysis process for framework development, called *Sherlock*. It is based on both FODA [KCH+90] and Proteus [HP94] domain analysis techniques. Proteus is composed of three iterative phases: domain description and qualification, domain requirements and architectural modeling, and

model validation. Similar to Proteus, Sherlock uses object-oriented modeling techniques, and is tightly related to a reference incremental software process (the waterfall model), in order to efficiently handle domain evolution according to the authors [VSF97]. The process also takes the complete documentation from FODA to prescribe how to model the requirements into reusable components of frameworks. The approach has been validated with the development of a GUI framework at an Italian software company [VSF97].

Johnson [JOHN93] points out that analyzing domain requires analyzing existing application, which is very hard and is expensive. Furthermore, it is only possible if those applications are available. Thus, advocates of top-down approaches argue that the cost can be balanced since a DSSA is rather stable, because “it is possible to predict the changes that are likely to be needed to a system over its lifetime” [WL99]. However, empirical studies have shown that software developers can only identify a subset of future changes and they cannot provide the complete picture of change [LS98]. Coplien et al. [CHW98] also recognize that it would be extremely difficult to predict unknown future requirements of products in a domain.

2.8.2.3 Hot Spot Generalization

Schmid [SCHM97] claims that it is not cost-effective to start framework design by trying to model its variability and flexibility at once. Instead, he suggests an “*generalization transformation*” approach that place an emphasis on hot spots, which are the driving force of framework design. The approach is summarized as follows:

1. Develop an object model of an individual application from the framework domain with standard object oriented analysis methods.
2. Identify the necessary variation points in the model and write the *hot spot specification*. A hot spot specification includes the description of required *variability*; *granularity*, whether the hot spot only covers one elementary variable aspect; *multiplicity*, the number of alternatives that may be bound to the hot spot; and *binding time* of the hot spot.
3. For each hot spot, design a hot spot subsystem, which is composed of an (abstract) base class, concrete derived subclasses, and possibly additional classes

and relationships. Hot spot subsystems are classified into three categories: *non-recursive* if a requested service is provided from one subclass object; *1:1 (chain-structured) recursive* if a requested service may be provided by multiple subclass objects that are structured in a chain; and *1:n (tree-structured) recursive* if a requested service may be provided by a tree of subclass objects. Schmid also categorizes the design patterns [GOF94] with the same classification and uses the patterns to realize the subsystems.

4. Associate a hot spot system to each identified variation point to generalize the application to a framework.

Free [FREE99] proposes a similar approach called the *hot-spot-driven development process*, which starts with an UML object model that captures the domain-specific requirements instead of an application object model. The object model is usually developed by domain experts and experienced software engineers with Class-Responsibility-Collaboration (CRC) cards [BC89]. A CRC card describes the name of a class, the responsibility of the class, and the relationships between the class and other classes. He also suggests using use cases and scenarios to identify and validate the object model. Hot spots are documented in the *hot-spot cards*, which contains the name, description of the required flexibility, and the realization of the hot spot. Demeyer et al. [DRMG99] introduce a variation to the approach which assigns a separate abstract class for each dimension of variability of a hot spot, that is, put the hook methods called by a template method to different classes in order to increase flexibility.

A framework must embody a theory of the problem domain as the result of domain analysis, whether explicit and formal, or hidden and informal [RJ97]. However, existing hot spot generalization approaches do not encompass precise guidelines to perform domain analysis, nor mapping guidelines from the analysis results to design [HIM01].

2.8.2.4 Use Case Driven

Jacobson et al. [JGJ97] argue that existing software processes lack concrete techniques to model reusable architecture and components, and precise guidelines to map requirements to design and implementation. They invent the Reuse-driven Software Engineering Business (RSEB), which is a systematic, use case driven reuse process

based on the UML notations for large-scale software reuse. RSEB has separated processes for Domain Engineering and Application Engineering. The domain engineering process plans all applications at once, and captures the requirements of the applications within use cases [JCJO92]. Commonality and variability are specified with generalization and variation points in use cases. The “4+1 View” [KRUC95] approach is adopted by RSEB to model architecture. RSEB places an emphasis on modeling variability and keeping the traceability links of the representation of variability that ranges from the analysis, design, and implementation models. Although RSEB focuses on variability realization, it does not include essential domain analysis techniques such as feature modeling. To address this issue, Griss et al. [GFA98] extend RSEB to FeatuRSEB by adding a feature model into RSEB, as the center repository to store features for re-users to develop reusable architecture (see Section 2.4.3).

D’Souza and Wills [DW98] propose the *Catalysis* approach, which is a component-based approach to develop software using UML and its extensions. Catalysis relatively focuses on business driven solutions [HIM01]. It uses three UML modeling concepts to define components and component interfaces. The concepts are summarized as follows:

- *Type*: A type models component interfaces by defining the external visible behaviour of objects. The UML class construct is used to specify the attributes and operations of a type. A type does not have implementations.
- *Collaboration*: A collaboration specifies how interactions between objects or components occur with types and *actions*. Actions are specific invocations of operations defined in a type. A collaboration is viewed as a *design unit*, which defines a design for certain services specified in the system requirements. Collaborations can be composed to define a more complex service or even an architecture.
- *Refinement*: A refinement specifies a relationship between elements of types, classes, and collaborations. It refers to the abstraction process to generalize types and collaborations. The concept is mainly used to define different levels of abstraction of collaborations.

The authors suggest using frameworks as building components at the design phase, where a framework is viewed as a pattern of model or code that can be applied to

different problems. However, Catalysis does not illustrate how to develop the design artefacts, and to integrate patterns into architecture.

John McGregor et al. [MMM99] introduce the *use case assortment* approach for the requirement analysis of framework development. The approach combines a set of modeling heuristics with an analysis technique to identify commonality and variability in the use cases of the applications in the framework domain. Use cases exhibit a same pattern are grouped together to form abstract use cases and abstract actors, i.e. the *assortment* process. The assorted use case models provide framework developers the same type of support as standard application use case models to application developers.

2.8.2.5 Discussion

Johnson [JOHN93] suggests an ideal way to develop frameworks, which is summarized as follows:

1. Analysis: Analyze the problem domain with existing applications.
2. Design: Construct abstraction that can be specialized to cover the applications.
3. Test: Test the framework by using it to build applications.

Although his proposition is closely related to the Bottom-Up approaches, it also has impact on other approaches. Here, we will identify the common elements of the existing approaches and summarize them in Table 2.

Framework design emphasizes the elicitation of required flexibility. Although all approaches include domain analysis implicitly or explicitly, few of them have clearly prescribed an effective way to identify and organize the result of domain analysis. Except the top-down approaches, others often choose either an object model (with the aid of use cases and scenarios) or use cases to model the requirements. However, non-functional requirements such as performance or implementation standard may not be modeled with use cases, due to the intrinsic “function-oriented” property of use cases. Furthermore, experiences have shown that readability of use case models may be decreased by incorporating variability into already complicated models [GFA98] [VAM+98].

Commonality	Bottom-Up	Top-Down	Hot Spot Generalization	Use Case Driven
Domain Analysis	Domain knowledge is obtained implicitly by building applications.	Domain analysis techniques such as feature modeling.	Develop a domain object model.	Capture the domain requirements into either use cases or feature models.
Variability Realization	Hot spot Design pattern	Hot spot Design pattern	Hot spot Design pattern	Hot spot Design pattern
Test	Yes	Yes	Yes	Yes
Iterative process	Yes	Yes	Yes	Yes

Table 2: Commonality of Framework Development Approaches

Traceability from requirements to design is essential to guarantee the realization of the required flexibility. Furthermore, developing a framework requires an iterative approach in which the framework is refined a number of times [BOOC94]. Thus, adequate supports for change propagation are expected to maintain the consistency between different artefacts of a framework. Although the Use Case Driven approaches stress traceability links among different models in order to guarantee the realization of variability, they do not provide a precise guideline to address the issue.

2.8.3 Framework Evolution

Johnson and Foote [JF88] claim that developing a Black-Box framework at the early stages of the framework’s history is extremely expensive and difficult. Most frameworks start their lifecycle as a White-Box framework, which makes heavy use of inheritance and the application developers must know how a component is implemented in order to reuse it. As a framework becomes more refined, it leads to “black box” components that can be reused without knowing their implementations. Ideally, each framework will evolve into a Black-Box framework.

Roberts and Johnson [RJ97] observe that a framework always evolves through a

number of levels of *maturity* as the framework developers increase their understanding of the framework domain and the required customization:

- White-Box: the application developers create subclasses in order to customize the framework and need to look at the code of the abstract classes.
- Component Library: many concrete subclasses are available for the application developers, much fewer subclasses are created during customization, compared with the White-Box level.
- Pluggable Object: extensive use of delegation and there are concrete subclasses available to serve as the targets of delegation, so the customization is mainly achieved by parameterizing subclasses.
- Fine-grained Object: the role or functionality of the abstract classes is decomposed into smaller classes, which allow more mix-and-matching of pluggable objects because of their finer granularity.
- Black-Box: application developers customize the framework without the knowledge of the internal design, instead, existing fine-grained components are selected and composed to build applications.
- Visual Builder: the choice of components and their composition is obtained by using drag-and-drop in a graphical interface.

The above maturity level describes a common path that a framework takes in evolution. However, there is no explicit description of where and how a framework evolves when it has reached a certain state [MB99].

Framework evolution comprises two dimensions: evolves as all software; and matures from initial versions to a stable platform [RJ97]. At each stage of maturity, the way that a framework is applied for the development of an application is different, and demands different documentation in order to make the job of the application developers easy [FSJ99].

Codenie et al. [CHSV97] observe that there is very little support for framework evolution. They identify a number of the most common problems regarding framework evolution, which are summarized as follows:

1. Structural complexity: Framework evolution can make its structure difficult to manage and comprehend.
2. Changes in the domain: A framework has to evolve once the framework domain changes. There are three approaches to attack this problem: define the original framework domain much wider than the current domain scope to accommodate possible future domain changes; redesign a new framework to cover both the original and the new domain; and reuse ideas from the original framework to develop a new framework for the new domain. Each of them has pros and cons. None of them has been widely accepted as an appropriate solution [FSJ99].
3. New design insights: Design of a framework may have to be enhanced because of issues that were previously neglected.
4. Evolution conflicts: Changes in a framework may cause the incompatibility between the architecture of the framework and the architecture of previously created applications.

Bosch [BOSC00] claims that the fundamental concern of using software product line is product line evolution. He [BMMB00] also states that the above issues have not been addressed by the existing framework development approaches. The first issue is software aging problem (see Section 2.1.2), which is caused by the inconsistency between different software artefacts during evolution. Traceability facilitates software comprehension and change propagation to improve software maintainability [JBR99]. However, the existing development approaches do not have adequate support to maintain traceability, as described in Section 2.8.2. One way to address aging problem is to document software precisely and completely, according to Parnas [PARN94].

2.8.4 Framework Documentation

Documentation is viewed as a key step in framework development [BCC+02]. Properly documenting a framework is important to facilitate its understanding and use. Framework design is very abstract and sometimes incomplete; and the collaboration and dependencies among classes can be indirect and obscure. Therefore, understanding a framework is more difficult than understanding a single application. On the

other hand, it should take substantially less time to understand and reuse a framework than to build an equivalent application without using the framework. Moreover, a framework is typically developed by expert designers who have profound domain knowledge and design experiences, whereas an application developer who reuses the framework might be less knowledgeable and less experienced. This kind of situation also stresses the criticality of good framework documentation [BD99].

The documentation of a framework has to address different audiences to meet their needs [BD99]. The audiences can be divided into four groups from the perspective of framework reuse:

1. Application developers: An application developer wants to know how to customize the framework to build applications. He is concerned about the relevant hot spots and how to customize them. An application developer may not be either a domain expert or an experienced developer.
2. Framework maintainer: A maintainer must understand the internal design of the framework, including its architecture, design rationale, class collaboration, problem domain, etc. A maintainer is both a domain expert and a software expert.
3. Framework verifier: A verifier must validate certain properties of the framework in order to meet rigorous customer requirements. He is mainly concerned about whether the documentation is clearly specified. A verifier can be either an application developer or a framework developer.
4. Developer of another framework: A framework developer seeks ideas from existing frameworks. He is concerned about the high level abstraction of the framework. He may not be a domain expert for the reusing framework.

Different audiences have different focuses and need different information [BD99]. Johnson [JOHN92] argues that framework documentation needs at least three parts: the purpose of the framework, to whom the documentation addresses; how to use the framework, which is the most important documentation part; and the design of the framework, which includes the detail design in terms of classes and their collaborations. Framework documentation that meet the criteria help framework developers construct clear concepts of the design and the steps required for customization. Hence,

documentation verifies whether a framework is easy to use, and this is the overriding goal of framework design [BCC+02].

Most framework users are not interested in the details of a framework but are looking for documentation that describes how to use the framework. A *cookbook* of a framework is a collection of *recipes*, each of which describes how to perform a typical example reuse of the framework [BD99].

Krasner and Pope [KRAS88] use a cookbook approach to describe the Model-View-Controller (MVC) framework. The documentation comprises an informal description of the framework design, the implementation detail, and a set of examples, which demonstrate how the framework can be used.

Pree [PREE94] introduces an *active* cookbook approach that describes an object oriented framework using *metapatterns*. He classifies the design patterns [GOF94] into three groups: patterns based on *abstract coupling*, the situation that a class has a reference to an abstract class; patterns based on recursive structures; and the remainder. Pree combines the concepts of abstract coupling and recursive structures with the notion of multiplicity to identify seven metapatterns for template and hook methods. Each variation point of a framework is described with a metapattern, which defines what the template method or class is, and which is the hook method or class. The framework design is put into a hypertext system, which is used as an active cookbook for the framework users. Schmid [SCHM97] has integrated the concept of metapattern into his generalization transformation approach for framework development.

Cookbook approaches document the purpose of the framework and present examples. The guidelines of how to use the framework are described informally in natural language. Most of them do not provide a precise mechanism to specify the detailed design of frameworks. Furthermore, the major weakness of cookbook approaches is that it only describes the typical way to use the framework, but lacks support for the unpredicted use of the framework [JOHN92].

Johnson [JOHN92] suggests using a set of patterns to document a framework. The format of a pattern is composed of three parts: a description of the problem; a detailed discussion of the alternatives to solve the problem, with examples and cross references; and a summary of the solution. The first pattern in the documentation describes the framework domain by giving examples, introduces the rest of the patterns, and often

specifies which patterns should be read next. He also places an emphasis on using examples. He has applied his approach in the documentation of HotDraw [BC87], a framework for semantic graphic editors.

Butler and Dénommée [BD99] state that the main audience group of framework documentation is the application developers, who may be inexperienced in either software development or the domain knowledge. They recommend a guideline on how to document a framework to assist application developers:

1. An overview of the framework setting a context for the domain and the variability in the framework. A simple example application may be used as the first recipe in the cookbook.
2. A set of example applications that are graded from simple through to advanced. Hot spots should be introduced incrementally. A complex hot spot may need multiple examples to illustrate its customization.
3. A cookbook of recipes are organized as Johnson's pattern language [JOHN92]. The recipes should use the example applications to make their discussion concrete. Cross references should be used between recipes, recipes and source code, and any other available documentation, such as design patterns.

Frameworks are more abstract than most software, which makes documenting them difficult [JOHN92]. Furthermore, deciding whether a framework is successfully well documented is also difficult, since there is no generally accepted documentation approach that covers all aspects of frameworks [BMMB00].

2.9 Refactoring

Software must continue to evolve to adapt to ever-changing requirements. One way to reduce evolution cost is to automate aspects of the evolutionary cycle whenever possible [ROBE99]. *Refactoring* is a behaviour-preserving program transformation that automatically updates an application's design and underlying source code [FOWL99]. In this section, we will at first introduce source code refactoring since it is the cornerstone of refactoring research. Part 2 gives a brief view on refactoring formalisms. Part 3 discusses the tool support for refactoring. Part 4 explores refactoring of other software artefacts. The last part describes the open issues of refactoring.

2.9.1 Program Refactoring

Opdyke [OPDY92] integrates Banerjee's approach [BKKK87], the design principles of Johnson and Foote [JF88], and the design history of UIUC Choices software system into object-oriented program refactoring. He categorizes software changes into three levels: high level requirement changes, low level source code changes, and the intermediate level between them. Opdyke introduces the term *refactoring* as "reorganization plans that support change at an intermediate level". For example, a refactoring that moves a member variable from one class to another one. He also identifies the intrinsic property of refactorings: refactorings should not change the *behaviour* of a program. Opdyke [OPDY92] suggests the use of *preconditions*, which are the context that a program must satisfy to apply a refactoring, in order to preserve the behaviour of programs. He defines twenty-three *primitive refactorings* and three *composite refactorings*. They are referred to as "low-level refactorings", in contrast to the three "high-level refactorings" that are supported by the former. Each low-level refactoring is defined rigorously with parameters, preconditions, and proof of behaviour preservation. They are given in the following list; the last three are the composite refactorings:

1. `create_empty_class`: defines a new class without members.
2. `create_member_variable`: defines an unreferenced member variable in a class.
3. `create_member_function`: defines a member function in a class that is either unreferenced or identical to an inherited function.
4. `delete_unreferenced_class`: removes an unreferenced class.
5. `delete_unreferenced_variable`: removes an unreferenced variable from a class.
6. `delete_member_functions`: removes a set of member functions from a class that are either redundant or unreferenced.
7. `change_class_name`: changes the name of a class.
8. `change_variable_name`: changes the name of a variable.
9. `change_member_function_name`: changes the name of a member function and all the inherited member functions in the subclasses.

10. `change_type`: changes the types of variables and the return types of functions.
11. `change_access_control_mode`: changes the visibility of a member variable or function.
12. `add_function_argument`: adds a new argument to a function and all the overriding functions in the subclasses.
13. `delete_function_argument`: removes an argument from a function and all the overriding functions in the subclasses.
14. `reorder_function_argument`: re-arranges the arguments in a function and all the overriding functions in the subclasses.
15. `add_function_body`: adds a function body to an existing function signature.
16. `delete_function_body`: deletes a function body from an existing function.
17. `convert_instance_variable_to_pointer`: converts the type of a variable from an instance of a class to a pointer type of that class. This refactoring is specific to C++.
18. `convert_variable_reference_to_function_calls`: converts all references or assignments to a variable to calls to its accessing or updating functions, respectively.
19. `replace_statement_list_with_function_call`: replaces a statement list with the function call to a function that carries out the same behaviour of the list.
20. `inline_function_call`: replaces a function call with the body of the called function.
21. `change_superclass`: changes the superclass of a class.
22. `move_member_variable_to_superclasses`: moves a variable to the superclass from all subclasses where the variable is defined.
23. `move_member_variable_to_subclasses`: moves a member variable from its current containing class to each of the immediate subclasses.
24. `abstract_access_to_member_variable`: defines a set of functions to replace all references to a variable with calls to those functions.

25. `convert_code_segment_to_function`: defines a new member function to replace a statement list that has the same behaviour to the function body.
26. `move_class`: migrates a class to a new location in its hierarchy.

Opdyke [OPDY92] gives examples of the three high-level refactorings in C++ programs and discusses their behaviour-preservation properties.

1. refactoring to generalize: creates an abstract superclass for multiple classes.
2. refactoring to specialize: decomposes a large complex class to several smaller classes and creates an inheritance hierarchy with them.
3. refactoring to capture aggregation and components: creates part-whole relationships between classes to improve flexibility.

He observes seven program properties, also called *invariants*, to preserve the behaviour of C++ source code refactoring:

1. Unique superclass: every class must have at most one superclass.
2. Distinct class names: each class must have a unique class name and classes cannot be nested
3. Distinct member names: all member variables and functions within a class must have distinct names
4. Inherited member variables not redefined: a subclass cannot redefine a member variable of its superclass.
5. Compatible signatures in member function redefinition: redefinition of a member function must have the same type signature of the overriding function.
6. Type-safe assignments: the type of each expression assigned to a variable must be an instance of the variable's defined type or subtype.
7. Semantically equivalent references and operations: changes can be made on variables that are either unreferenced, or the new references are semantically equivalent to the old ones after refactorings

Finally, Opdyke implies that his refactorings do not apply to programs that are dependent on the size or physical layout of objects.

Many of the primitive refactorings defined by Opdyke are implemented by Roberts in his Smalltalk refactoring browser [RBJ97]. Roberts [ROBE99] extends Opdyke's definition of refactoring by adding *postconditions*, which are assertions that a program must satisfy after the refactoring has been applied. The idea comes from the observation that refactorings are typically applied in sequences which set up preconditions for later refactorings. He argues that using postconditions can reduce the amount of analysis that later refactorings have to perform, derive the preconditions of composite refactorings, and calculate dependencies between refactorings. The preconditions of a composite refactoring are deduced via sequentially evaluating the preconditions of each refactoring in the interpretation that has been transformed by its earlier refactorings in the composite refactoring. He claims that since large design changes can be composed by a sequence of smaller, primitive refactorings, the entire composition is also a refactoring [ROBE99]. The dependency between refactorings is defined in terms of *commutativity*, which refers to the assumption that refactorings do not have to be performed in the sequence in which they are supposed under certain conditions. Although he proposes a formula to calculate the conditions under which any two refactorings may commute, the issue of how to determine the correct refactoring order still exists. Roberts has also examined run-time analysis techniques that assist dynamic refactoring in programs.

Tokuda [TOKU99] implements Opdyke's refactorings in C++, and has shown that those invariants cannot always preserve the behaviour of transformed C++ source programs. He proposes additional refactorings to support design patterns as target states for software restructuring efforts [TB99]. Class structures are changed to an applicable design pattern to increase flexibility. Tokuda [TB95] claims that at least seven patterns from the "Gang of Four" book: Abstract Factory, Adapter, Bridge, Builder, Strategy, Template Method, and Visitor, can be viewed as the target of a program transformation [GOF94]. He argues that the architectural changes of object-oriented systems can be classified into three types [TOK99]:

1. Schema transformations: the schema of an object-oriented database management system is similar to the class diagrams of an object-oriented application
2. Introduction of design patterns as micro-architectures: a number of patterns

can be viewed as automated program transformations that are applied to an evolving design.

3. Hotspot generalization: refactorings on identified hot spots can be automated through simple framework creation and adding necessary design patterns

He has shown that all three types of changes can be automated by refactorings [TOK99].

Tokuda [TOK99] views a refactoring as a parameterised behaviour-preserving program transformation, and suggests the following template to describe refactorings:

- *Purpose*: the reason why the refactoring is performed
- *Arguments*: the entities involved in the refactoring
- *Description*: how to perform the refactoring
- *Enabling conditions*: under which circumstance can the refactoring be performed
- *Initial state and target state*: the structure of the involved entities before and after the refactoring

Fowler [FOWL99] explains the principles and best practices of refactorings. A comprehensive catalogue of seventy-two proven refactorings is presented in the book. Each refactoring has a name, a short summary, a motivation that describes why the refactoring should be applied, a step-by-step description of how to apply the refactoring, and an example in the Java programming language. There are no conditions in a refactoring to be satisfied to ensure behaviour preservation. Instead, he places an emphasis on testing and suggests one prepare a set of test cases prior to start a refactoring. The test cases are executed against the changed program after each step of the refactoring. His book is a landmark in making refactoring known to programmers in general.

Refactoring is gaining more and more recognition through the application of *eXtreme Programming* (XP) processes [BECK99]. In XP, developers evolve the design incrementally upon new requirements from the clients. One of the key aspects of XP is continual refactoring of the source code. Many object-oriented Integrated Development Environments (IDE) provide considerable support for XP, such

as Eclipse [STOR02], an extensible open source IDE that supports two core activities in the XP process: refactoring and unit testing.

An intuitive and pragmatic way to check behaviour preservation of program refactoring is to run an extensive set of test cases and compare the pre- and post-refactoring results. However, Pipka [PIPK02] argues that tests relying on the program structure, which may be modified by the refactorings, may show different results. Thus, it is essential to perform behaviour-oriented unit tests before and after refactorings. Large refactoring processes should be decomposed to set up checkpoints that reveal missing or overseen preconditions to preserve the system's behaviour.

2.9.2 Refactoring Formalisms

Refactoring can be represented as graph transformations. Heckel [HECK95] claims that a direct correspondence between refactorings and graph transformations exists. Programs or even other kinds of software artefacts can be specified with graphs. The nodes are the software entities such as classes, variables and methods, while the relationships between those entities such as inheritance, variable accesses and method calls are represented by edges between the corresponding nodes. Refactorings correspond to graph production rules and the application of a refactoring corresponds to a graph transformation.

Mens et al. [MDJ02] present the formalisation of refactoring using graph rewriting, a transformation that takes an initial graph as input and transforms it into a result graph. This transformation occurs according to a set of predetermined rules that are specified in a graph production, which is specified by means of a *left-hand side* (LHS) and a *right-hand side* (RHS). The LHS is used to specify which parts of the initial graph should be transformed, while the RHS specifies the result after the transformation. The “semantics” of a program is defined with well-formedness constraints, which are specified with *type graphs*, a meta-graph expressing restrictions on the instance graphs that are allowed; and *forbidden subgraphs*. A graph is well-formed only if there is a graph morphism into a type graph, that is, all node and edges can be mapped to the type graph and the mappings preserves sources, targets and labels. Forbidden graphs exclude illegal configurations in a graph, so that a graph satisfies the constraints expressed by a forbidden graph if there does not exist a morphism between the graph and the forbidden graph. The preserved “behaviour” of refactoring

is expressed with the implementation of each method that is involved in the refactoring. The “behaviour” can be categorized into: *access*, if each method implementation accesses at least the same variables before and after the refactoring; *update*, a method updates at least the same variables before and after the refactoring; and *call*, if each method implementation performs at least the same method calls before and after the refactoring. The approach has been validated with Fujaba, a graph-rewriting tool that is tightly integrated with Java and UML [MEJD04].

Banerjee et al. [BKKK87] investigate object oriented database schema evolution and identify a set of invariant properties of an object-oriented schema that must be preserved during schema changes. Changes are specified with a set of transformations rules. However, there are no rules that allow changing the location of a method in a class hierarchy. Their work is recognized as the origin of object-oriented software refactoring since object-oriented database schemas can be seen as the predecessor of UML class diagrams [MT04].

Mens and D’Hondt [MD00] propose an evolution contract formalism to manage UML model transformation. They introduce the concept of *evolution contract* as the formal constraints between the *provider* and the *modifier* of an artefact with well-formedness rules expressed in the Object Constraint Language (OCL) [OMG03]. The provider clause of an artefact specifies what properties the artefact can be depended on, while the modifier clause describes how to change the artefact precisely. The UML metamodel is extended to incorporate the evolution contract as a subclass of the Dependency metaclass. The exact “semantics” of model transformations is specified with *contract types*, which are defined as the stereotypes of the evolution contract metaclass. A *primitive* contract type performs the creation or deletion of model elements and relationships, while a *composite* contract type is composed of primitive types. Unexpected behaviour alteration or evolution conflicts can be detected by comparing evolution contracts during model transformations. They claim that the formalism can deal with the evolution of all kinds of UML models since it is defined at the metamodel level.

The Design Maintenance System (DMS) [BAXT92] is a rule-based transformation system. It works with a hierarchy of domains, each of which is specified with syntax, semantics, and mappings to the same or other domains. DMS can implement source code transformations, and has been used for transformation of COBOL programs for

the removal of duplicate code and dead code. However, the DMS transformations cannot ensure behaviour preservation.

Philipps and Rumpe [PR97] present a calculus for stepwise refinement of abstract data flow architecture in terms of components and connectors (also referred as *channels*). Architecture is rendered as a network which is composed of boxes (components) and arrows (connectors). The calculus consists of a set of graph transformation rules, such as adding or deleting components and channels. The behaviour of a component is modeled as a relation from the set of input channels of the component, to the set of output channels. A whole system is viewed as a black-box component and its behaviour is specified with the composition of the behaviour of all the components that are included in the system. The correctness of rules is justified by refinement relations on the black-box views of architecture.

Griswold [GRIS91] suggests using meaning-preserving transformations to restructure programs written in a functional programming language, called Scheme. Many transformations he chooses are compiler optimization techniques such as function extraction. He uses *program dependency graphs* to reason the correctness of transformation rules, in order to ensure the “semantics” preservation. Those rules are not well suited for object oriented languages since they are concerned with functional languages and do not consider things like inheritance. He also observes that class hierarchies complicate transformations.

Lieberherr et al. [LHR88] introduce a programming language independent rule, called the *Law of Demeter*, which organizes and reduces the behavioural dependency between classes to guarantee that methods have limited knowledge of the object model. The law is originated from work with the Demeter system, which provides a high level interface to object-oriented systems. They have shown that any object-oriented program written in “bad style” can be systematically transformed into an equivalent program that obeys the law [LH89]. The class hierarchy in Demeter is described using production rules. A collection of these rules is called the *class dictionary*. The transformation algorithm is defined with the data structure called a *class dictionary graph*, in which classes are represented as the nodes, and their relationships are the edges.

It is sensible to use graph transformation as the formalism to specify refactorings

because graphs are a language-independent representation of the source code. Moreover, transformation rules precisely specify code transformation, and the formalism allows the proof of behaviour preservation. Nonetheless, it is extremely complicated to deal with large nested structure with graph transformations, and the behaviour preservation still cannot be guaranteed [MT04].

2.9.3 Tool Support

Although it is possible to execute refactorings manually, tool support is considered crucial. Automate refactorings reduce the cost and tedium of debugging and testing commonly performed modifications, which would otherwise have to be performed manually, to the internal structure of software systems [BECK99].

A refactoring process can be viewed as a sequence of activities, each of which can be automated to certain extent [MT04]:

1. Identify the software artefact that should be refactored
2. Determine which refactorings should be applied
3. Verify the behaviour preservation of the applied refactoring
4. Apply the refactoring
5. Evaluate the consequence of the refactoring
6. Propagate the changes to other artefacts to maintain traceability

As Mens [MD03] points out, contemporary refactoring tools usually only support the automation of step 4, and neglect the remaining steps. Thus, they are referred as “semi-automatic” tools since the developers still have to manually identify which part of the software needs to be refactored, and select the most appropriate refactoring to apply. However, Tokuda and Batory [TB01] argue that even a semi-automatic approach can significantly improve the productivity in terms of coding and debugging time, compared with manual refactoring.

Tourwé and Mens [TM03] propose a logic meta programming (LMP) based approach to devise a refactoring tool that automates step 2. They use SOUL, a Prolog-like logic programming language that is implemented on top of Smalltalk, as the

metalanguage. All entities and relationships in source files can be directly accessed from the SOUL environment through a metalevel interface of *representational mapping predicates*, which are used in queries to retrieve matching entities for refactoring opportunities. Once an opportunity is found, a list of applicable refactorings is presented. They also observe the fact that the application of an individual refactoring may open possibilities for other refactorings to be applied. They call this phenomenon *cascaded refactoring opportunities*, which can also be automatically detected. The identification of refactoring opportunity, choice of applicable refactorings, and cascading refactorings are based on predefined logic rules. They have integrated the approach into a Refactoring Browser in the VisualWorks Smalltalk IDE.

Astel [ASTE02] suggests using an UML tool to analyze the refactoring possibility in source code, and perform elaborate refactorings via direct manipulation on the class diagrams of the code. He claims that many people prefer to visualize the classes and their relationships, and refactorings at design level may be more efficient than code refactorings in certain situations, such as when a class being changed involves multiple source files. Refactorings can be done by simple drag-and-drop actions. He argues that it is necessary to have a reverse-engineering tool to automatically generate diagrams from code, and to keep the code and models synchronized.

Boger et al. [BSF02] present a refactoring browser for UML to detect evolution *conflicts* that may be introduced as the side effects of refactorings. There are two types of conflicts: *warning*, which indicates a possible side effect; and *error*, which will harm the model or source code. They claim that most refactoring tools for UML only apply to static structure in terms of class diagrams and lack support to handle changes on dynamic behaviour. Thus, their tool aims at state and activity diagrams. Refactorings are expressed with state merging, decomposing, parallelization, and sequentialization.

Mens [MENS05] observes that no existing tool that provides adequate support for refactoring of software artefacts other than source code, and thus may cause consistency problem of the involved artefacts during software evolution.

2.9.4 Other Refactorings

Software is composed of many different types of artefacts range from analysis to design, implementation and test. Thus, all these artefacts should be kept consistent

when any of them is being refactored during software evolution. While many techniques are available for program refactoring, some researchers shift their focus from source code to other software artefacts.

The concept of an evolution contract [MD00] is originated from the *reuse contract*, which is suggested by Steyaert et al. [SLMD96] to handle change propagation between inheritance class hierarchy during software evolution. A reuse contract is an interface that is composed of a set of method description, each of them consisting of a name, an annotation, and a specialisation clause that lists the methods required by this method. Evolution is specified with *reuse operators*, which define the transformation rules on class hierarchies. Mens [MENS01] extends their work to allow arbitrarily complex reuse contracts, in order to handle UML collaboration with graph rewriting rules.

Judson et al. [JCF03] describe a metamodeling approach to perform *pattern-based model refactoring*, which incorporates appropriate design patterns into UML design models for perfective evolution. Transformations are specified with an extended UML metamodel that is composed of: a *source pattern*, the classifier of source models to which the transformations can be applied; a *target pattern*, the classifier of target models; and a *transformation pattern*, which characterizes the transformations to integrate a design pattern. A transformation pattern includes a *transformation schema*, which identifies the model elements that are created and deleted by the transformation; and a *transformation constraint* that stipulates the preserved relationships between the target and source model elements.

Garg et al. [GCC+03] present a graphical environment, called *Ménage*, to manage the evolution of software product line architectures. Ménage uses a XML-based Architecture Description Language (xADL2.0) [WH02] to describe product line architectures in terms of components, connectors, and their interfaces with *structure and types* schemas. All elements (including interfaces) are typed, and the schema also supports the specification of sub-architectures to address scalability in architectural specification. Optional and variation points are expressed with Boolean expressions. Configuration of an architecture can be achieved by applying a set of user-specified criteria, which is automatically interpreted by Ménage.

Critchlow et al. [CDCH03] suggest an approach to enhance the Ménage environment with two tools. They propose the *service utilization metrics*, which address the

structural quality of product line architecture. There are two types of metrics for a component: the percentage of provided services that are actually used by other components, and the percentage of required services that are actually provided by other components. A diagnostic tool, called ARCHMETRIC, is used to automatically calculate and visually present the metrics of all components of a product line architecture in each configuration. The results are analyzed manually to obtain the instructions for the refactoring tool, called ARCHREFACTOR, which automates a set of predefined refactorings on the architecture based on those instructions. The refactorings are mainly concerned with changes to variability, such as changing an optional component to be a core component.

Back [BACK02] introduces a *stepwise feature introduction method* for software construction that is based on incrementally extending the system with a new feature at a time. The behaviour of a system must be preserved when a new feature is added, or when the system structure is changed to fit new features. Software architecture is viewed as a hierarchy of layers. Behaviour preservation is handled with *correctness conditions* in the refinement calculus. Each class is assumed to have a class invariant, which expresses the conditions of the attributes that must be established when the class is instantiated, and which must be preserved by each operation in the class. A method can be called only when its preconditions is satisfied, and possibly post-conditions, which express the invariants when the calls return. Back argues that his approach is a complement to XP because XP literature does not precisely define how to structure software in an incremental way.

Russo et al. [RB98] suggest restructuring natural language requirement specifications by decomposing them into a structure of *viewpoints*, each of which represents partial requirements of system components. They state that refactorings increase the comprehension of requirements and detect inconsistencies.

2.9.5 Open Issues

Mens and Deursen [MD03] identify emerging trends in refactoring research, and observe a list of open issues, which are summarized as follows:

1. Identify the appropriate formalisms and techniques for different kinds of refactorings

2. Analyze and manage dependencies between refactorings that are performed together
3. Extend the notion of behaviour from functionality to include other software quality attributes
4. Address the scalability of refactorings to handle large industry setting software
5. Build extensible refactoring tools for user-specific or domain-specific customization
6. Maintain traceability between software artefacts at different abstraction levels during refactoring

They also state that the last issue also applies in a framework based or product line software development process, when applications are built from framework instantiation. When a framework is refactored, different instantiations may become inconsistent. Thus, adequate support to manage consistency in framework refactoring is desirable.

2.10 Software Traceability

A software system does not only include its code, but also its documents or other artefacts, as the specification of the system in different levels of abstraction. Each artefact is viewed as a model of the system [LIND94]. *Traceability* is the ability to trace the dependent items within a model and the ability to trace the corresponding items in other related models [PB90]. It is used to understand the entire software process and its artefacts, to know exactly the relationship between each requirement and its corresponding design and implementation, also to verify whether the requirement is implemented [RG93]. Complex software systems evolve continuously to meet changing user needs. Keeping traceability during software evolution propagates the changes on requirements to design and implementation, and provides a way to find out how those parts are affected by the changes [DW98]. Thus, traceability facilitates software comprehension and change propagation to improve software maintainability [JBR99].

Finkelstein [FINK91] explains the importance of traceability with another way. He argues that there will be no need of traceability if software can be developed with formal methodologies and the implementation is completely generated from a formal specification. Jacobson [JACO87] introduces the concept of seamlessness. A software development methodology is *seamless* if a software system can be completely generated from its specification and no seam between the two models of the system, with the methodology. On the other hand, there will be *semantic gaps* between models, if the development methodology is not seamless. Two models are seamlessly related to each other if concepts introduced in one of the models can be found in the other model through a simple mapping. The mapping is based on the *traceability schema*, which are rules to map concepts in one model to concepts in other models. Jacobson et al. [JCJO92] claim that object oriented methods are better than structured methodology since the former result in smaller semantic gaps between models than the latter.

2.10.1 Requirements Traceability

The most common use of the term traceability is requirements traceability. *Requirements traceability* is “the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of ongoing refinement and iteration in any of these phases)” [GF94]. Backward traceability is concerned with whether each requirement explicitly refers to its source in previous documents, while forward traceability is the ability to refer to each requirement in the requirements specification in design and implementation. Requirements traceability allows software maintainers to keep track of the change in the design and implementation due to the requirement alteration during software evolution. It can also assist the justifications and decision making in the design and implementation phase.

Requirements traceability of software systems is important to different stakeholders, such as clients, project managers, analysts, designers, testers, maintenance personnel, and end-users [RE93]. They view traceability from various perspectives in terms of goals and priorities. Requirements traceability is intended to ensure continued alignment between stakeholders’ requirements and the software artefacts in the

development life cycle [RJ01]. The source and rationale of requirements should be documented in order to understand requirements evolution and verification.

Empirical studies show that even experienced software developers can not predict all possible future change of a software system [LS98]. On the other hand, more accurate costs and schedules of changes can be determined with complete traceability, rather than depending on the developers to know all the areas affected by these changes [RJ01]. The estimation can be aided by *impact analysis*. It is the process to analyze which parts of a system are affected by proposed change, and how much they are affected [PB90]. Impact analysis is one of the main application areas of traceability [KELL90]. Requirement verification can also benefit from traceability [KELL90].

Gotel and Finkelstein [GF94] classify requirements traceability into two kinds: pre-RS traceability and post-RS traceability. The *pre-RS traceability* is concerned with the aspects of a requirement's life prior to its inclusion in the requirements specification (RS); and the *post-RS traceability* is concerned with the aspects of a requirement's life that derived from its inclusion in the RS. Standard software engineering approaches mainly address post-RS traceability. Research has shown that most of the requirements traceability related problems are caused by the lack of pre-RS traceability [FINK91] [GF94]. Finkelstein [FINK91] has argued that pre-RS traceability problem will exist since it is inherently paradigm-independent. In addition, he also mentions the obstacle of tracing non-functional requirements, which were not treated by most software development methodologies, and hard to be verified.

Ramesh and Jarke [RJ01] describe a set of requirements traceability reference models based on empirical studies. The participants were different types of stakeholders, and categorized into low-end and high-end groups with respect to their traceability practise. There are separate reference models for the two groups, respectively. The low-end traceability users view traceability as the documents transformation of requirements to design. The main applications of traceability are requirements decomposition, requirements allocation into system design, and change management. The high-end users of traceability employ much richer traceability schemes than low-end users. The main applications of traceability cover full life cycle including the communication of clients and end users, design issues and rationales.

Knethen [KNET02] introduces a conceptual trace model based on the analysis of change-oriented traceability for embedded systems. The trace model determines the

types of traceable documentation entities and relationships to support impact analysis and change propagation. The trace model consists of conceptual system model and conceptual documentation model. A *conceptual system model* describes logical entity types, their dependency and refinement relationships that are included in the system at different abstraction levels. Logical entity types and their relationships depend on the investigated application domain. A *conceptual documentation model* describes representation entity types and their relationships that are included in the documents at various abstraction levels. A conceptual documentation model of a system can be derived from the conceptual system model.

2.10.2 Dimensions of Traceability

Given a software system, *horizontal traceability* is the possibility to trace a requirement throughout the models, i.e. from requirements throughout the design to its code implementation. Horizontal traceability conforms to the definition of requirements traceability. *Vertical traceability* is the possibility to trace dependencies within a model [PB90]. It is also termed as tracing by *hierarchy*.

Traceable models should include requirements, specifications, and implementations [RE93]. Traceability from requirements to design can be recorded into *design rationale*. It refers to why a certain design decision is made and which requirements is fulfilled by a certain decision [CONK89].

Horizontal traceability and vertical traceability form a *traceability web*, which is depicted in a directed graph shown in Figure 12 [LS96]. It is based on Pfleeger's work [PB90]. The nodes represent the requirement, design component, and code, and the edges specify the link of traceable dependency, also called *traceability link*. Every node in a model may have many edges to nodes in other models. Lindvall [LIND94] argues that decreasing the complexity of the tracing dependencies assists impact analysis and increases the system maintainability.

Traceability can be distinguished to be explicit traceability and implicit traceability. *Explicit traceability* is the ability to trace via predefined traceability links or other kinds of references between items in two models. *Implicit traceability* occurs in all situations where explicit traceability does not exist [LIND94].

Level (*granularity*) of traceability refers to the level of traceable detail between

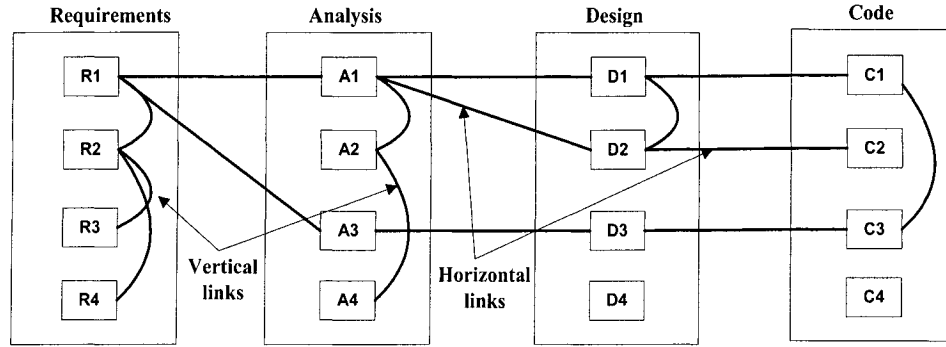


Figure 12: The Graphical Traceability Web

models [LIND94]. The coarsest level is the ability to trace from a document to another document, and the most refined level is to trace every single statement. Smith et al. [SDKD03] define granularity as the size of the sub-units of an artefact under consideration in the traceability context. Coarse granularity implies that an artefact has few sub-units, while fine granularity indicates a relatively large number of sub-units. They propose a framework to facilitate change propagation automation [SDKD03]. Change propagation is regarded as two dimensions: the automation degree to implement change propagation, and the automation degree to be notified of change propagation. Given an artefact that is involved in change propagation, its *traceability value* is the product of these two dimensions and the granularity at which the artefact is located.

2.10.3 Traceability Characterization

Lindvall [LIND94] provides a set of concrete traceability examples from the perspective of a software maintainer, based on the case study on a large scale commercial project using Objectory [JCJO92] as a methodology for system development. He characterizes traceability into a two-dimensional classification based upon the examples.

The first dimension is the traced items:

1. Object-to-object traceability: an object in a model is mapped to a corresponding object in another model.
2. Association-to-association traceability: the ability to trace an association in one

model to its corresponding association in another model. If an association cannot be directly mapped to the range model with the same semantics, discussion, ratification, and verification should be performed to maintain the traceability.

3. Use-case-to-object traceability: the mapping is based on the object participation of a use case.
4. Use-case-to-use-case traceability: the mapping is based on the functional requirements specified by a use case.
5. Two-dimensional object-to-object traceability: it deals with tracing class inheritance hierarchy between two models. Inheritance represents dependencies within a model (vertical traceability) since the descendant is depended on all its ancestors. On the other hand, the traceability links represent horizontal traceability.

The second dimension is concerned with the tracing process:

1. Tracing via explicit links: the standard process of a typical development environment.
2. Tracing by using references: textual references are used to trace between different documents.
3. Name tracing: items are traced with names. It is based upon the assumption that a consistent naming scheme is applied during the model construction.
4. Concept tracing: experienced developers use in-depth system and domain knowledge to trace interrelated items.

He argues that the characterization is not only useful for maintainers to understand the systems, but also beneficial to the cost estimation of change impact analysis.

Olsson and Grundy [OG02] introduce an approach to support traceability and change propagation between functional and non-functional requirement description, use case models and Black-Box test plans. The metamodels of the requirement model, use case model, and test case model are defined in terms of context, data, and interaction. The model elements are captured from the corresponding representation such as high-level natural language requirement documents, and summarized into

pre-defined forms. Vertical traceability information is inherently specified by the metamodels. The horizontal inter-model relationships are classified into:

Exact: Exact duplication information between two models such as name tracing.

Similar: Similar concepts between two models.

Generalisation: More abstract information in one model based on information in another model.

Specialisation: More detailed information in one model based on information in another model.

Multiplicity: The multiplicity information of elements between two models. The relationship is refined into Splits (1: many), Merges (many: 1), and Exact group (m: n).

2.10.4 Automation and Tool Support

It is essential to have automation tools to support traceability management since creation and validation of trace information has been proven to be extremely time consuming and error-prone [EGYE01].

Lange and Nakamura [LN97] present an approach to obtain, manipulate, and visualize runtime traces that provides fine-grained localization information for program understanding. Object interactions are captured into an object interaction graph based on the observable behaviour during execution. Search space reducing techniques such as merging and pruning are applied to remove unwanted information from the graph, thus limiting its complexity and size. The research prototype, called Program Explorer, takes a user through a series of executions and visualizations to comprehend a given program, even to investigate instances of design patterns in a system.

Olsson and Grundy [OG02] have prototyped an information management tool to support their approach. The tool includes a set of “extract agents”, which are used to capture information from requirements, use cases, and test cases. The information is organized and presented with web user interface for users to navigate and manage the traceability links.

Egyed [EGYE01] describes an iterative approach to generate traceability information based on observing test scenarios at runtime, to tackle with the consistency problem of legacy software reverse engineering. He refers to the source code that is executed while testing a scenario as *footprint* of the scenario. The approach consists of four major activities: *Hypothesizing* analyzes existing documents and models to extract the hypothesized traceability information; the information is organized into a footprint graph during *atomizing*; the graph is traversed from each leave node to their parents, as *generalizing*, and from the root to its leaves as *refining*, to further explore the traceability dependencies of the nodes. A tool, called TraceAnalyzer, is developed to support the automation of the approach, other than the Hypothesizing activity.

Zisman et al. [ZSPK03] propose an approach that automates the discovery of bi-directional traceability relationships between requirements artefacts. The approach focuses on three specific types of documents: a commercial requirements specification (CRS), a functional requirements specification (FRS), and a requirements object model (ROM). Both the CRS and FRS are expressed in the XML language, while the ROM is described in a UML class diagram. The process is driven by the ROM. It starts with finding traceability relationships between both the CRS and ROM and the FRS and ROM. These relationships are then used to create the relationships between the CRS and FRS. They have developed a prototype to automate the process.

Chapter 3

Cascaded Refactoring Methodology

We never know the worth of water
till the well is dry.
~Anonymous

Standard software development methodologies are not sufficient for developing frameworks [PG94]. Methodologies for the development of a framework have been suggested that use domain analysis, iterative design, software evolution, and design patterns [JOHN93] [PG94] [TALI95] [FSJ99]. However, there are no mature framework development methodologies [FHLS99]. Having carefully examined them, we found three main issues [BUTL99] [BX01] (see Section 1.1.3):

1. Identification and realization of required variability for the family of applications
2. Framework evolution
3. Framework documentation

The Cascaded Refactoring Methodology addresses these problems and provides a moderate solution. The methodology chooses a set of models to describe a framework: a feature model which identifies and organizes the domain commonality and variability; a use case model that captures the requirements; an architectural model to specify the high level collaboration of layers and subsystems; a design model that illustrates the interactions of classes; and the framework source code. A framework

is developed in an iterative process, i.e. evolution. The evolution is naturally viewed as refactoring followed by extension, the same as in those Bottom-Up approaches for framework development that are centered on refactoring [RJ97]. In the cascaded refactoring methodology, the overall refactoring of a framework is regarded as a set of refactorings performed sequentially on the models, and the constraints on how to refactor a particular model are determined by the refactorings on the previous models. Hence, the reconstruction cascades from one model to the next. Alignment maps are defined to maintain the consistency and traceability of models during the refactoring. As part of the documentation, the rationale, choices, and impacts of refactorings are recorded with a template defined in the methodology.

The remainder of this chapter is organized as follows. Section 3.1 describes meta-models of the chosen models, and specifies the model notations. Section 3.2 defines the alignment maps amongst the models. Section 3.3 elaborates the methodology.

3.1 Models

A metamodel is a precise definition of the constructs and rules needed for creating semantic models [OMG03]. It is the collection of concepts with which a certain domain can be described. Metamodels are typically built upon a strict ruleset to specify the definition. In most cases, the ruleset is derived from entity-relationship diagrams or object-oriented class diagrams.

Metamodels of the feature model and the architectural model are defined. The literature of those models have been investigated, elements and relationships organized into the metamodels in UML notations. We have referred to the metamodel definition in UML [RATU03] and Siemens approach [HNS99]. The basic structure of a metamodel is:

- *Syntax* : elements and relationships of the metamodel are presented in UML class diagrams
- *Semantics*: the definition of the elements and relationships are described in the natural language
- *Justification*: the explanation of the metamodel in natural language

The metamodels are not studied as formal languages. The correctness and completeness are discussed informally in the justification part. The use case metamodel proposed by Rui [RB03] is adopted in the methodology. The metamodels are used as the basis to describe refactorings and to justify preservation of behaviour.

3.1.1 Choice of Models

The design of a framework is similar to the design of most reusable software [TRAC95]. It starts with domain analysis to identify the requirements, constructs a generic design to meet the requirements, and implements the solution in target languages. Then the framework is used to build applications to verify its flexibility and extensibility. The verification finds weak points and leads to design improvements. At the time during the analysis, design, and implementation, different models can be chosen to specify the framework. Here, we will explain the rationale behind the choice of models in the methodology.

3.1.1.1 Feature Model

A framework is always the result of domain analysis, since the development of a framework demands efficient ways to capture the requirements in terms of commonality and variability [RJ97]. Furthermore, it is more important to identify the variability than to identify the commonalities since the goal of developing a framework is to provide a reusable platform, which supports sufficient flexibility and extensibility [ARAN94] [FSJ99].

Feature modeling analyzes commonalities and variability among a family of related applications in terms of features, and organizes the analysis results into a feature model [KCH+90]. The construction of feature diagrams can help framework developers identify the variation points of the framework. Since the introduction of Feature Oriented Domain Analysis (FODA) approach in 1990, many domain engineering and software product line methods have adopted feature concept to specify the commonality and variability in domain analysis [KLL+02]. Furthermore, Bosch and Gibson argues that features can be viewed as units of incrementation as applications evolve since the differences between the products of a product line can be specified in terms of features [GIBS97] [BOSC00]. As an analysis technique, the feature modeling should be done very early in the development process. Features “should be the first class

objects in software development” [KKL+98]. Domain analysis with feature modeling also provides good support when specifying use cases [JCJO92]. Thus, a feature model is chosen to identify and capture the commonality and variability of a framework, and is chosen as the starting point of cascaded refactorings.

Although use case models can also capture commonality and variability [MMM99] [BRJ99], feature models cannot be neglected for three reasons [GFA98]. First, feature models are “re-user” oriented and use case models are typically user oriented. Second, a use case model emphasizes “what” applications in the domain do, whereas a feature model specifies which functionality can be selected when an application is being built. Third, use cases gather and describe user requirements in terms of functionality, whereas feature models organize commonality and variability in a much broader range and cover all aspects of domain concepts. Moreover, experience has shown that use case models can be very complicated with variation points to express extensibility in some domains, thus impeding the development [GFA98] [VAM+98].

3.1.1.2 Use Case Model

Use cases capture the functional requirements of a target system, as it is meant to behave in the host system [JCJO92]. A use case describes how a user will make use of the system in a time-sequential order [KK03]. The use the user makes is modeled by the passing of signals or information between the user and the target system.

Use case modeling has become a standard approach of requirement analysis and specification, especially after its integration into UML [JBR99]. Many use case driven framework development approaches involve use case models [JGJ97] [MMM99], even feature model based domain engineering approaches such as FeaturSEB [GFA98]. Thus, a use case model is chosen to fill the gap between framework requirements in terms of features and the framework design artefacts. Use cases support the transition from a black-box view of a software system, i.e. the interactions between the system and external entities, to a white-box view of the system, which not only models the external interactions but also specifies the internal structure and actions of the system [BCKR97]. Use cases relate the operations of the target system to their design and implementation in terms of calls between subsystems. This can be viewed as the starting point of system decomposition and the subsystem interface design. In addition, scenarios derived from the use cases of a framework can

be used to validate its design prior to having the complete implementation of the framework [CP95] [MMM99].

The hierarchical view of use cases is provided by the Generalization, Inclusion, and Extension relationship of use cases [BRJ99] [OMG03]. Abstract use cases may occur in a use case model to express commonality amongst use cases, even if an abstract use case has no concrete scenario. The Inclusion relationship connects a task with a subtask which is also represented as a use case. Extension points allow variability in services to be specified. Therefore, use cases are capable of modeling commonality and variability with those relationships [BRJ99].

3.1.1.3 Architectural Model

Software must have a solid foundation at the level of software architecture [PW92]. Framework documentation should include the architectural design of the framework [JOHN92]. Software architecture is the high-level abstraction of the components and their collaboration of the software system [SG96]. The architecture of a framework organizes the structure of the framework in terms of layers and subsystems, distributes the responsibilities to their interfaces, and realizes the required flexibility through hot spots and design patterns [SCHM97] [DW98] [FSJ99].

In addition, it is possible to take a use case view of a given subsystem, where the rest of the system in the architecture is regarded as actors that request the services from the given subsystem [BCKR97]. Thus, the services of the subsystem are described in its use case model, and accessed through its interfaces. It is also possible to take a class view of a subsystem, where one identifies a subsystem with a facade class, and the subsystem interfaces are identified with the class methods [GOF94] [OMG03]. Hence, an architectural model can aid the traceability from the requirements to the design of a framework.

3.1.1.4 Design Model

A design model is chosen to illustrate the detail design of frameworks. The UML structural and behavioural models are adopted since UML has become the de-facto standard in software modeling language [BRJ99] [KK03] [OMG03]. The design model of a framework represents the static structure and dynamic behaviour of the framework. The design pattern usage is encouraged because design patterns provide the

flexibility and extensibility of the framework [JOHN92] [GOF94] [BJ94]. Design patterns are also effective in framework documentation [JOHN92] [BD99]. Using commonly known design patterns helps framework developers understand the framework by serving as a common vocabulary between the framework builders and the application developers [GOF94].

3.1.1.5 Source Code

The source code of a framework is part of the framework documentation [BD99]. It describes how the requirements are implemented, how the objects are organized, how to provide the flexibility and extensibility with hooks and templates in the target programming language. The source code is also indispensable for the framework verification.

3.1.2 Feature Model

A feature model provides an overview of the requirements of a software product line in terms of commonality and variability [CN02]. It is used for the derivation of the customer's desired product and provides a hierarchical structure of features according to the decisions associated to them [KCH+90] [KKL+98].

3.1.2.1 Metamodel

Although much work has been done on feature modeling since the last decade, there is no consensus on the explicit semantic of feature models [KCH+90] [KKL+98] [RBSP02]. On the other hand, it is essential to have clear and coherent feature model semantics to support its refactoring. Thus, we have defined a metamodel. The metamodel (Figure 13) incorporates the elements and relationships of feature models from various research works in the literature, and focuses on FORM. The syntax is described in the UML class diagram notation.

A feature model consists of features and the relationships between them. A *Feature* is a view of a concept, which usually represents a property of a software product line. In a feature diagram, the concept is represented as the root feature. A *Feature* has the following attributes:

- *name*: each feature has a unique name of string type.

- *kind*: it is Enum type because features are classified into four categories in FORM:
 - *Capability features* are externally visible behaviours or the way users may interact with the product, such as services, operations and performances.
 - *Operating Environment features* represent attributes of the environment in which the product operate. They describe the product's conformance to standards or external entities, such as hardware interfaces or constraints.
 - *Domain Technology features* are types of requirement decisions to develop domain models, such as terminology, domain specific methods.
 - *Implementation Technique features* represent low-level implementation issues, such as the C++ Standard Template Library.
- *isRoot*: indicates whether the feature is the root feature. It is a Boolean value.

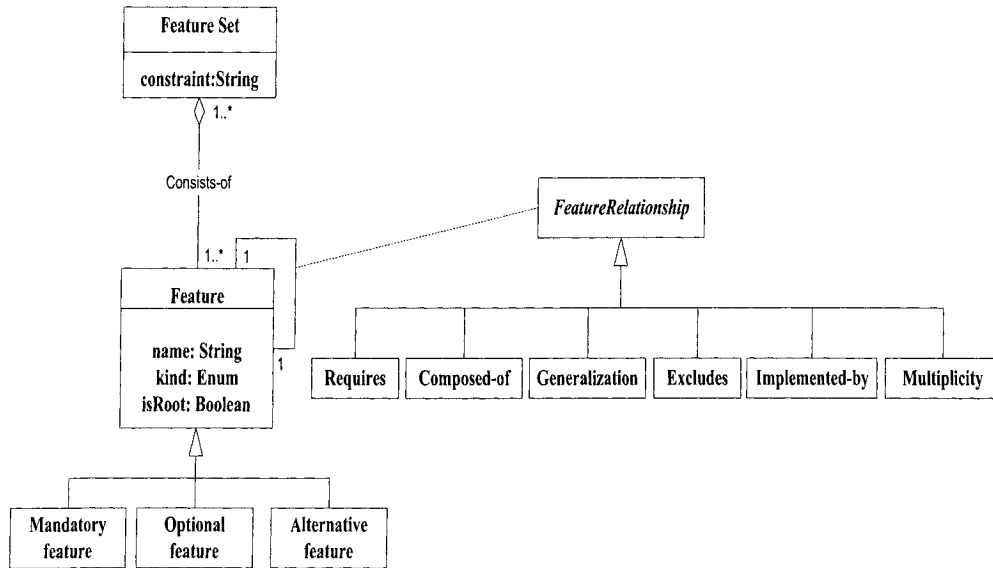


Figure 13: Feature Model Metamodel

In terms of variability, features can be classified into mandatory, optional and alternative. A *Mandatory feature* specifies a core property of the concerned domain concept, and is common to all instances of the concept. An *Optional feature* represents a property that may not be necessary to some instances of the concept. An *Alternative feature* represents different ways to configure a mandatory or optional feature.

Features are organized with *FeatureRelationships* in a feature model. *FeatureRelationship* is an abstract metaclass. There are six specific types of feature relationship in the metamodel. A “*Composed-of*” relationship means a feature can be a composite of its sub-features. A “*Generalization*” relationship indicates that one feature is an abstraction of another feature. Similar to generalization in the object-oriented paradigm, a parent feature can be replaced by a child feature at the location in which the parent feature is used. An “*Implemented-by*” relationship means a feature is a prerequisite to implement another feature. A “*Multiplicity*” relationship is used to specifies allowed number of occurrence of a feature. It is represented as m..n (m and n are non-negative integers specifying the lower and upper bound on the number of occurrence.) A “*Requires*” relationship indicates that to have one feature in a feature set, another feature must also be included. An “*Excludes*” relationship between two features means that they cannot be simultaneously selected in a feature set.

A *Feature Set* consists of a finite set of features and the constraints over the features. An instance of the product line can be specified by a valid Feature Set, which satisfies all the constraints. A Feature Set has the following attributes:

- *constraint*: the rules which should be satisfied by the features included in the feature set. We only integrate a limited set of the generic constraints in a feature set:
 - If the super-feature of a mandatory feature is included in a feature set, the mandatory feature must be included in the set, too.
 - If a super-feature of an optional feature is included in a set, the optional feature may or may not be included in the feature set.
 - Only one alternative feature is included in the feature set to which the super-feature of the alternative feature belongs.
 - If feature A has a Requires relationship with feature B, feature B must be included into any feature set which contains feature A.
 - If feature A has an Excludes relationship with feature B, feature B must not be included into any feature set which contains feature A.

3.1.2.2 Metamodel Justification

The metamodel is not defined as a formal language, and cannot be validated with theoretical reasoning. We justify our choice of metamodel by comparing its models to those in the literature. A summary of existing feature models is shown in Table 3.

	Feature Relationships	Feature Types
FODA	Aggregation Requires (dependency) Mutex-with (dependency)	Mandatory Optional Alternative
FeatuRSEB	Composed-of Requires (dependency) Excludes (dependency)	Mandatory Optional Alternative XOR OR
Generative Programming	Aggregation Requires (dependency) Excludes (dependency)	Mandatory Optional Alternative OR
FORM	Composed-of Generalization Implemented-by Mutual dependency (dependency) Mutual exclusion (dependency)	Mandatory Optional Alternative
Riebisch's Extension	Aggregation Refinement Multiplicity Requires (dependency) Excludes (dependency)	Mandatory Optional Alternative
Metamodel	Composed-of Generalization Implemented-by Multiplicity Requires Excludes	Mandatory Optional Alternative

Table 3: The Summary of Feature Model Elements and Relationships

There has already been consensus on the conceptual definitions of Feature and Feature Sets [KCH+90] [JGJ97] [GFA98] [CE00]. A Feature can be viewed as a requirement of the software products in a domain [KCH+90]. However, the “scope” of

a Feature is much broader than traditional software requirements. A Feature cannot only represent technical aspects of a software product, but also the managerial aspects, such as deliverable deadline, budget cost, etc. Even for the technical aspects alone, Features can represent functional requirements such as operations, services, and non-functional requirements such as performance, reliability, robustness, portability, etc. It is difficult to construct accurate maps from a feature model to other models such as a use case model or a design model, with such a broad feature definition. Therefore, we followed FORM and defined the *kind* attribute in the Feature element [KKL+98]. The functional requirements that are represented by Capability features are very close to the services described in use case models. The Operating Environment features specify the conformity to interfaces of external software and hardware products. They can be related to the interfaces in the architecture. The Domain Technology features describe the domain concepts, which can be mapped into design models. The Implementation Technique features can be mapped to the source code constructs.

The metamodel only divides features into mandatory, optional, and alternative in terms of variability. However, the semantics of other type of features are also covered by the metamodel. For example, the same semantics of “or” feature can be obtained by combining the Multiplicity relationship with Alternative features [CE00] [RBSP02].

A comprehensive set of feature relationships are encompassed in the metamodel. It covers FODA, FORM, FeaturSEB, Generative Programming, and the latest Riebisch’s extension. The Mutual Reliance and Mutual Exclusion feature dependencies are defined as Requires and Excludes relationship in the metamodel. The Aggregation relationship in FODA, Generative Programming, and Riebisch’s extension specifies the same concept as the Composed-of relationship in FORM and FeaturSEB. It is represented as the Composed-of relationship in the metamodel. The Refinement relationship defined in Riebisch’s extension means one feature can be specialized by other features. It is same as the Generalization relationship in FORM. Thus, they are defined as the Generalization relationship in the metamodel. The Implemented-by relationship in FORM and the Multiplicity relationship in Riebisch’s extension are incorporated into the metamodel, as well.

3.1.2.3 Metamodel Subset

We only use a subset of the feature metamodel in our case study of the cascaded refactoring methodology. The subset is given in Figure 14.

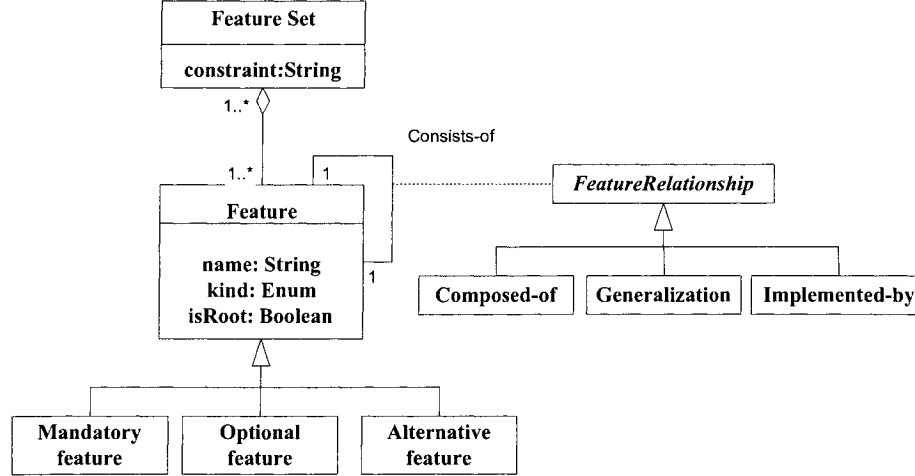


Figure 14: Feature Model Metamodel Subset

3.1.3 Use Case Model

The use case metamodel defined by Rui and Butler [RB03] is adopted in the methodology. It incorporates Regnell's use case specification [REGN99] and other works in use case modeling [JCJO92] [RAB96] [FHG98].

3.1.3.1 Metamodel

The metamodel (Figure 15) consists of three level views: the *Environment Level* shows the relationships between use cases and the external entities; the *Structure Level* describes the use case internal structure; and the *Event Level* represents the lower abstraction level in terms of events.

A *Use Case* represents a system usage that is characteristic to a specific Actor. An *Actor* defines a coherent set of roles by which users of the system can perform during their interactions with the system. A *User* is an instance of an actor. One User can be the instance of multiple Actors in different contexts. External systems or devices which communicate with the target system can also be modeled as Actors.

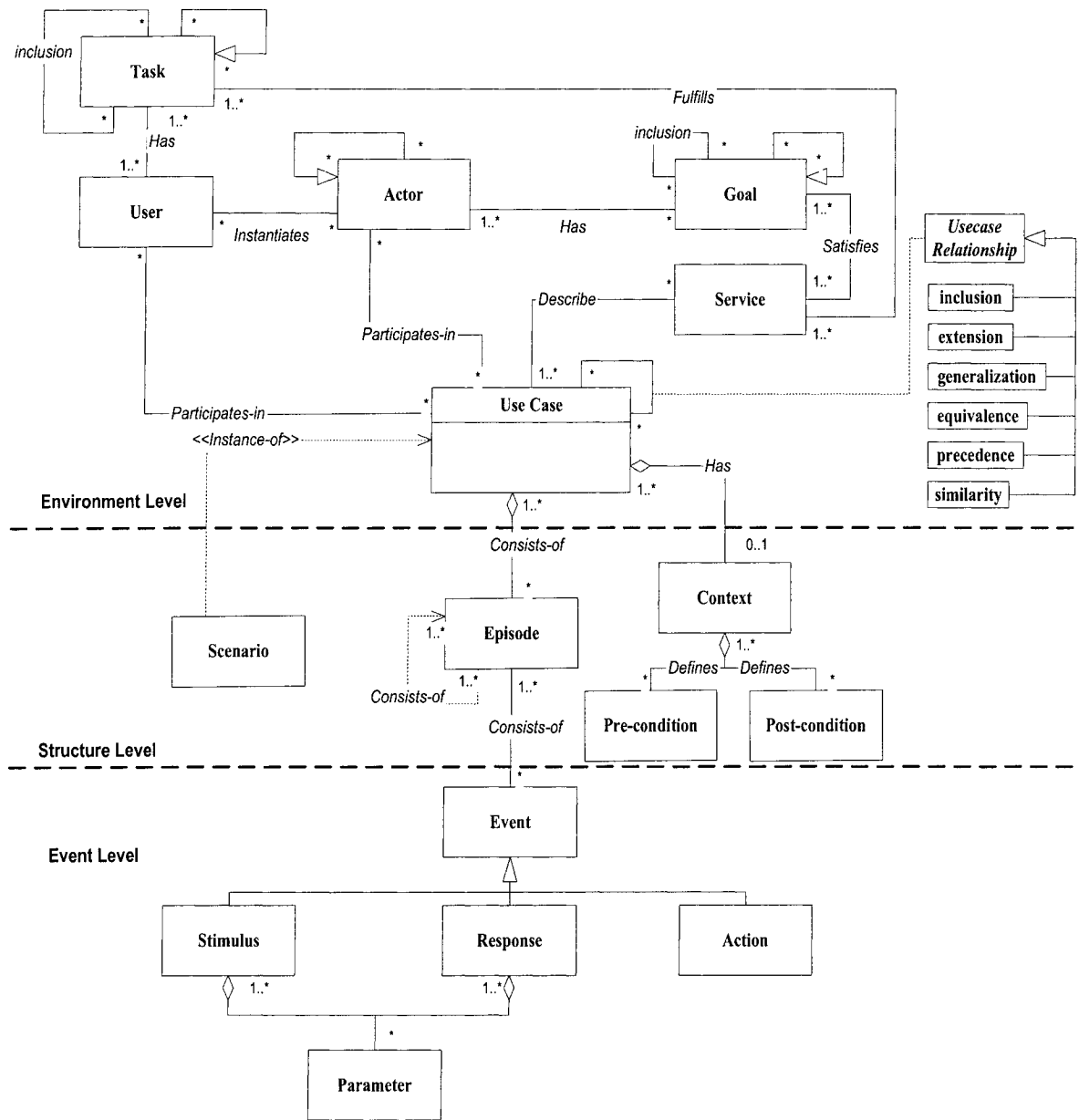


Figure 15: Use Case Metamodel

Actors may have commonalities; i.e. communicate with the same set of Use Cases in the same way. The commonality is expressed with the *generalization* relationship between Actors. An instance of a child Actor can be placed where an instance of the parent is expected. Use Cases describe *Services* that are provided by the target system to its Users to accomplish Goals. A *Goal* is an objective of Users when they request Services. An *inclusion* relationship between two Goals specifies that one goal includes another goal as a sub-goal. Multiple Goals may share commonalities, i.e. contain the same set of sub-goals. This is represented by the *generalization* relationship between goals. A *Task* represents the way in which the Users interact with the system. An *inclusion* relationship between two Tasks indicates that one Task includes another one as a subtask. Multiple tasks may have commonalities, i.e. contain the same set of subtasks. This is described by the *generalization* relationship of Tasks.

A *Scenario* illustrates a specific realization of a Use Case as a sequence of a finite number of events with linear time order. A Use Case can be initialized to a collection of Scenarios. A Use Case may be divided into coherent parts, called *Episodes*. An Episode represents a sub-task. One Episode can be included in multiple Use Cases. The *Context* of a Use Case defines its Pre-conditions and Post-conditions. A *Pre-condition* of a Use Case is the enabling states of the environment and the target system to execute the use case. A *Post-condition* is the state of the environment and the target system after the execution of the Use Case.

An Episode may consist of Events. An *Event* specifies a significant occurrence that has a location in time and space. An Event can be a Stimulus, or a Response, or an Action. A *Stimulus* is the passing of information from the Users to the target system. A *Response* is the passing of information from the target system to the Users. Both of them can take *Parameters* to carry data to and from the target system. An *Action* describes the intrinsic event of the target system. It is the atomic unit of a Use Case. There is no communication between the target system and the Users that take part in a Use Case during the execution of an Action of the Use Case.

Use cases are organized with *Usecase Relationships* in a use case model. The Usecase Relationship is an abstract metaclass. There are six specific types of use case relationship in the metamodel. An *inclusion* relationship between two Use Cases means that the behaviour defined in the target Use Case is included at some location in the sequence of behaviour performed by the base Use Case. When an instance

of a base Use Case reaches the location, it performs all the behaviour described by the included Use Case and continues its own event flow afterwards. An *extension* relationship means that a Use Case may be augmented with additional behaviour which is defined in another Use Case. The relationship contains a condition as an extension point in the base Use Case. The condition must be satisfied if the extension is to take place, and the extension point defines the location in the base Use Case where the additions are to be made. A *generalization* relationship between Use Cases implies that the child Use Case contains all the attributes, sequences of behaviour, and extension points defined in the parent Use Case, and participates in all relationships of the parent Use Case. The child Use Case may define a new sequence of behaviours, or specialize an existing behaviour of its parent. A *similarity* relationship between two Use Cases means that one Use Case resembles another one in some unspecified way. This relationship provides a way to foresee relationships among Use Cases even when the exact relationship is not clear yet. An *equivalence* relationship between two Use Cases specifies that one Use Case is equivalent to another one, i.e. an alias. A *precedence* relationship between two Use Cases defines that the behaviour of one Use Case is appended to that of the preceding one.

3.1.3.2 Metamodel Subset

We do not discuss the justification of the metamodel. Interested readers can refer to the original paper [RB03]. A subset of the use case metamodel (Figure 16) is used in the case study of the cascaded refactoring methodology. It covers the main concept and relationships of use case models defined in UML [BRJ99] [OMG03].

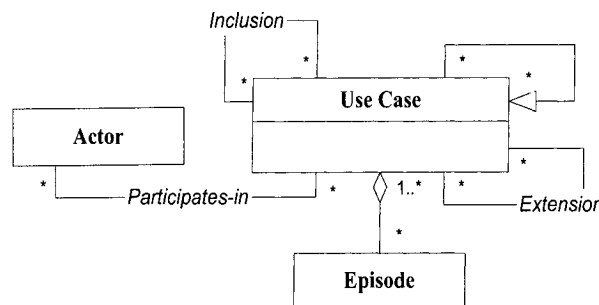


Figure 16: Use Case Model Metamodel Subset

3.1.4 Architectural Model

No consensus has been made on the architectural model of frameworks, nor does an metamodel exist [KRUC95] [SG96] [DW98] [RATU03]. Our architectural metamodel is a combination of the logical architecture view of the UML[OMG03] and the module architecture view of the Siemens approach [HNS99].

3.1.4.1 Metamodel

The syntax of the metamodel is depicted in the UML class diagram (Figure 17). All architectural elements have a common attribute:

- *name*: an unique name of string type.

An *Interface* is a named set of Operations that characterize the behaviour of the architectural element which realizes the Interface. The behaviour is characterized as the “service” role, while the element which provides the interface can be viewed as the “provider”. Interfaces are the connectivity points of other elements.

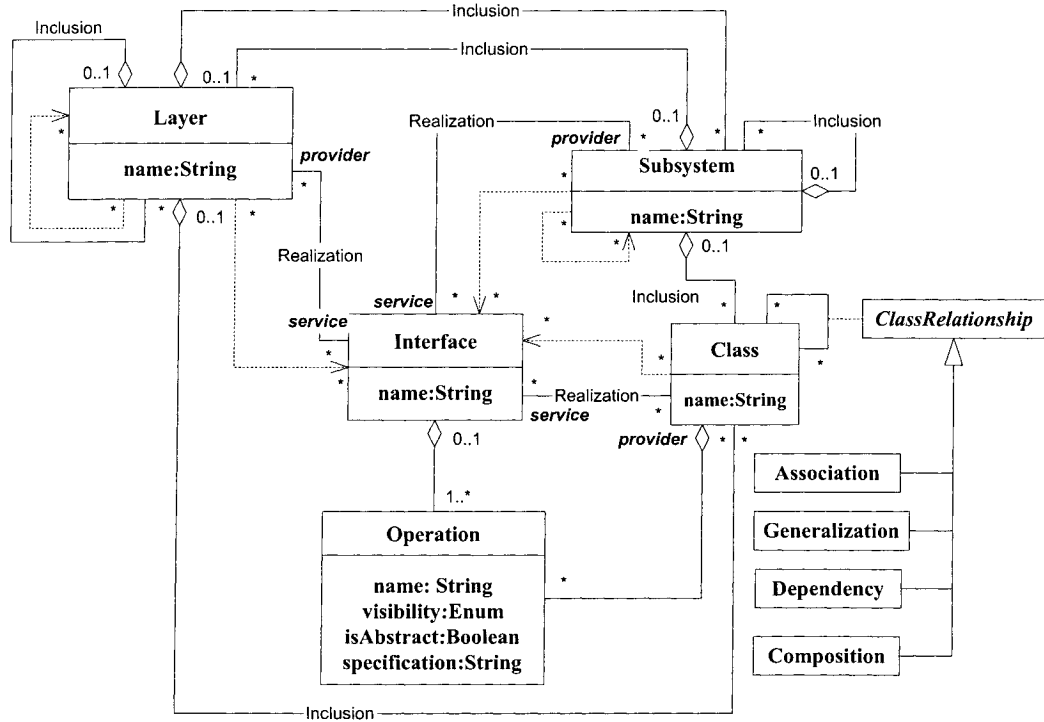


Figure 17: Architecture Metamodel

A *Class* represents a modeled concept that has data structure, behaviour, and relationship to others. The name of a Class has the scope of the Subsystem or Layer in which it is assigned. Classes interact with each other through Interfaces. A Class can realize and depend on multiple Interfaces. The relationships between Classes are represented by the abstract class *ClassRelationship*. It can be specialized to four subclasses. A *Generalization* relationship between two classes indicates that the more specific Class (child) inherits the structure and behaviour of the more general Class (parent). Moreover, additional properties can be added to the child Class. A child can be used anywhere its parent appears. An *Association* is a structure relationship which specifies that, an instance of Class A is always connected to an instance of Class B with a navigation path, if A has an Association relationship with B. If a Class requires an Interface, that is realized by another Class, to provide its services, there is a *Dependency* relationship between them. A *Composition* relationship between two Classes indicates that one Class (component) is a part of the other Class (composite). An instance of the composite always has instances of the component.

An *Operation* is a service that can be requested to effect behaviour. An Operation has the following attributes:

- *name*: each operation has a unique name of string type.
- *visibility*: it is enumerator type. The visibility attribute specifies whether the operation can be used by other elements. An Operation can have one of the following visibility values:
 - *public*: the Operation can be used by any external element with visibility to the element which contains the operation.
 - *protected*: the Operation can be used by any descendent of the element which contains the operation.
 - *private*: the Operation can only be used by the element which contains the Operation.
- *isAbstract*: it is Boolean type and used to indicate whether the Operation is abstract.
- *specification*: the signature of the Operation.

Classes with tight coupling relationship can be grouped together into a Subsystem. A *Subsystem* offers Interfaces to provide a collection of services. For each Operation in an Interface offered by a Subsystem, the Subsystem itself or at least one of its contained element must have a matching Operation. A Subsystem can realize or depend on multiple Interfaces. If a Subsystem requires an Interface which is realized by another Subsystem, there is a *Dependency* relationship between them. The *Inclusion* relationship indicates that a Subsystem can contain other Subsystems, Layers, and Classes. A Class cannot directly communicate with Classes that are external to its Subsystems; it has to interact through the Interfaces of the Subsystems. Classes can be assigned to a *Layer*. Layers provide Interfaces. If a Layer requires an Interface which is realized by another Layer, there is a *Dependency* relationship between them. The *Inclusion* relationship means that a Layer can contain other Layers, Subsystems, and Classes. Architectural entities that are included in a layer have to interact with entities outside the layer through the layer interface. Layers and Subsystems organize architecture into a partially ordered hierarchy in order to reduce complexity and flexibility.

3.1.4.2 Metamodel Justification

In addition to the Siemens approach [HNS99], there are two other well-known architectural modeling approaches: architectural styles, and the “4+1” views [SG96] [KRUC95] [DW98]. An *architectural style* is described in terms of components and connector types; a set of configuration rules, which constrain how components and connectors may be configured; semantic interpretation, which defines when suitably configured designs have a well-defined meaning as architecture; and analyses that may be performed on well-defined designs [SG96]. Architecture styles are not integrated into the metamodel because the concepts are covered by the Conceptual view in the Siemens approach.

The “4+1” view comprises four views plus a use case view to describe architecture [KRUC95]. It consists of a Logical view, a Process view, a Physical view, and a Development view. These views are carried over into the UML system modeling [BRJ99]. In addition, the main concept of the “4+1” view are covered in the Siemens approach except the use case view. The Siemens approach is preferred to the “4+1” view, because the description of the metamodels together with the stereotypes

and notations that extend UML make the Siemens approach much more precise than that of the “4+1” view. The Use Case view in the “4+1” view specifies the requirements in scenarios to validate the design. Its task can be taken over by the use case model in the cascaded refactoring methodology.

	Architectural Elements	Relationships
“4+1” View Development View	Layer Subsystem Module	Dependency Include
Siemens Approach Logical View	Layer Subsystem Module Interface	Contain Composition Use Require Provide Assigned to
UML Logical View	Package Subsystem Interface	Dependency
The Metamodel	Layer Subsystem Class Interface Operation	Dependency Realization Inclusion Composition Association Generalization

Table 4: The Summary of Architectural Model Elements and Relationships

In the Siemens approach, software architecture is described by four views. The Conceptual view presents the configuration of components and connectors. The Module view shows the software structure in terms of layer, subsystem, module, and interface. The Execution view identifies the hardware resources, communication mechanisms, and the runtime entities such as processes. The Code view presents the organization of source code, libraries, and executables. Our initial ambition was to integrate the four views into the architectural metamodel, but scaled back to only the Logical view due to limitation of architecture experiences. Besides, the case study is a single thread, standalone system running on a single processor. The Execution view of the architecture is trivial and not much different from the Module view [HNS99].

The Code view describes how the software implementation matches the design decisions in other views. The major work of the Code view is carried out by the source code related alignment maps in the methodology. The Logical view is prone to the solution domain as the refinement of the Conceptual view, which is more closely tied to the problem domain.

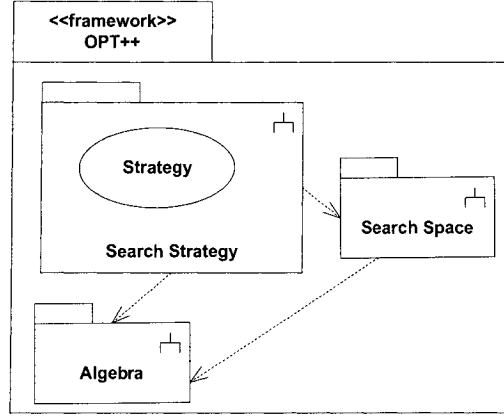


Figure 18: An Example of the UML Framework Concept

The UML architectural model also includes a Logical view in terms of package, subsystem, class, interface and their relationships [BRJ99] [OMG03]. UML defines a *framework* concept [OMG03] as an architectural pattern that provides an extensible template for applications within a domain. A framework is modeled as a stereotyped package. It provides a set of elements including classes, interfaces, collaborations, use cases, and even other frameworks. We have not adopted the framework notion into the architectural model because it does not provide additional mechanisms to assist the illustration of framework architecture, compared with the package concept in UML. An example is shown in Figure 18. The OPT++ is an optimizer framework for database management systems. It can be customized to support different optimization strategies. The framework is composed of three subsystems. The Search Strategy subsystem provides the strategies that are used to explore the search space. The Strategy design pattern supports the variation on the search algorithms. The Search Space subsystem defines what the search space is, which is determined by the way the query plan is structured and transformed. The Algebra subsystem defines the relational operators used in the DBMS and their corresponding algorithms to execute these operators. The Search Strategy subsystem depends on the Algebra and

the Search Space subsystems. The Search Space subsystem depends on the Algebra subsystem.

Table 4 summarizes the elements and relationships in the Development view of the “4+1” view approach, the Logical View of the Siemens approach and UML, and the metamodel. A Module in the “4+1” view is a collection of data and operations, and provides services through its interfaces. It has the same semantics as that in the Siemens approach. A Module can be mapped to an Ada package, a set of classes, a procedure, etc. Since we are only concerned about with the object oriented paradigm, we use the Class concept instead of Module. The Include relationship in the “4+1” view, the Contain and Assigned to relationship in the Siemens approach deliver the same meaning of nesting structure, which is represented as the Inclusion relationship in the metamodel. The Require and Use relationships in the Siemens approach describe the dependency between the consumers and the suppliers of services. Therefore, they are specified as the Dependency relationship. The Provide relationship in the Siemens approach represents the realization of interfaces, which is defined as the Realization relationship. A Composition relationship between Modules means that a Module can be decomposed into one or more Modules. It specifies the same concept as that of the Inclusion relationship in the metamodel. The Operation concept is defined as the quantified units of services. It facilitates the trace map between the architectural model and the design model, since its semantics is the subset of that of the Operation concept in UML [OMG03].

3.1.5 Design Model

UML has been proven to be the industry standard of modeling languages for software engineering practices [DW98] [BRJ99] [HNS99] [LL01] [OMG03]. It is used for specifying, visualizing, constructing, and documenting the artefacts of software systems. The UML structural and behavioural models are adopted to construct the design model in the cascaded refactoring methodology. Structural models emphasize the static organization of objects in a system, including their classes, interfaces, attributes and relationships. Behavioural models concern the behaviour of objects in a system, including their methods, interactions, collaborations, and state histories. Table 5 and Table 6 gives the elements and relationships of the design model [OMG03].

Design Components	Semantics
Class	A description of a set of objects that share the same attributes, operations, methods, relationships, and semantics.
Interface	A named set of operations that characterize the behaviour of an element and defines a service offered by the element.
Attribute	A feature within a classifier that describes a range of values that instances of the classifier may hold.
Operation	A service that can be requested from an object to effect behaviour. The service is described as a signature with a name and parameters.
Visibility	An enumeration whose value (public, protected, or private) denotes how the model element to which it refers may be seen outside in its enclosing namespace.
Abstract Class	A class that cannot be directly instantiated.
Object	An entity with a well-defined boundary and identity that encapsulates state and behaviour.
Type	A domain of objects together with operations applicable to the objects without defining the physical implementation of those objects.
Template	The descriptor for a class with one or more unbound parameters.
Collaboration	A set of participants communicates in a specific way to accomplish a task.
Message	A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue.
Constraint	A semantic condition or restriction.
Event	A specification of a type of observable occurrence.
Role	The named specific behaviour of an entity participating in a particular context.
Transition	A relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
State	The named specific behaviour of an entity participating in a particular context. A situation during the lifetime of an object when it satisfies certain condition, or waits for events.

Table 5: Design Model Elements

Relationships	Semantics
Generalization	A subtyping relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element in terms of properties, members, and relationships and may contain additional information.
Association	A semantic relationship connects between two classifiers.
Composition	A part-whole relationship between a composite and its components.
Dependency	A semantic relationship that indicates a situation in which a change to the target element may require a change to the source element in the dependency.
Realization	A semantic relationship that indicates that an interface is supported by a class (hierarchy).
Multiplicity	The range of allowable cardinalities that a set may assume.

Table 6: Design Model Relationships

3.1.6 Source Code

The cascaded refactoring methodology focuses on the development of object oriented application frameworks with object-oriented programming languages, such as C++. Mechanisms to support variability are encouraged in framework implementation, such as inheritance, polymorphism, templates, pre-processor, parameterization, etc.

3.1.7 Model Notation

It is important to have a set of pre-defined notations for each of the models, as the vocabulary to construct the graphic models. Here, we will define the model notations used in the case study of the methodology.

Figure 19 presents the notations of the subset of the feature model metamodel.

The UML notation standard [OMG03] is followed for the use case modeling. The Episode notation is defined, since it is not provided by UML. The use case model notation is shown in Figure 20.

The architectural model notations are given in Figure 21 and Figure 22.

The UML notation standard [OMG03] is followed for the design model, shown in



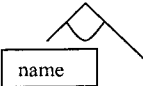



Element/ Relationship	Notation	Description
Mandatory feature		A rectangle with the feature name.
Optional feature		An empty circle is attached on the top of the rectangle. The feature name is inside the rectangle.
Alternative feature		Multiple lines begin from the super-feature, and end at the alternative sub-features. The lines are connected with an arc.
Composed-of relationship		A solid line connects the two features.
Generalization relationship		A dashed line connects the two features.
Implemented-by relationship		A feature is a pre-requisite to implement another feature. They are connected with a dashed line.
Feature Set	$\{f_1, f_2, \dots, f_n\}$	f_i ($1 \leq i \leq n$) are the names of the features in the feature set.

Figure 19: Feature Model Notation



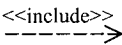
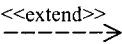


Element / Relationship	Notation	Description
Use Case		A use case is depicted as an ellipse, within which a unique text string is put inside as the name of the use case.
Actor		An actor is depicted as a stick man figure with the name of the actor below the figure
Episode	Episode name: "description of the episode"	An episode is depicted as a natural language statement of the task specified by the episode. The episode statement is placed inside the ellipse of the use case to which the episode belongs.
Inclusion		An include relationship is depicted as a dashed line with an open arrowhead from the base use case to the included use case. The arrow is labeled with the keyword <<include>>.
Extension		An extension relationship is shown by a dashed arrow with an open arrowhead from the extending use case to the base use case. The arrow is labeled with the keyword <<extend>>.
Generalization		A generalization relationship is depicted as a solid directed line with a large open arrowhead pointing to the parent use case.
Participates-in		A participates-in is symbolized as a straight line with an open arrowhead pointing from the actor to the use case.

Figure 20: Use Case Model Notation

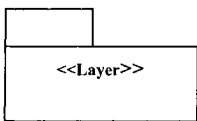
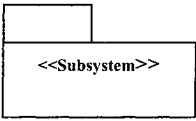
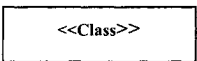
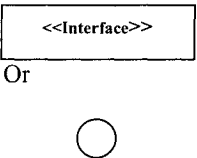
Element	Notation	Description
Layer		A partially ordered hierarchy of classes. A Layer always has a unique name.
Subsystem		A group of tightly coupled classes that collaborate together to provide services. A Subsystem always has a unique name.
Class		A Class is a modeled concept that has data structure, behavior, and relationships to others. A Class always has a unique name.
Interface		An Interface is a collection of Operations. The rectangle notion is chosen when the Operations of the Interface are presented. An Interface always has a unique name.

Figure 21: Architectural Model Element Notation






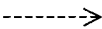
Relationship	Notation	Description
Composition		A Composition relationship between two Classes indicates that one Class is a part of the other one. The filled diamond is attached to the composite.
Generalization		A Generalization relationship between two Classes indicates that the child class is fully consistent with the parent class. The triangle is attached to the parent class.
Association		A solid line that connects to the two classes that has an Association relationship.
Realization	 or 	The line notation is chosen if the corresponding interface is depicted as a circle; otherwise the dashed line with a hollow arrow is used.
Dependency		The arrow begins with the depending element, and ends at the depended element
Inclusion	Nesting	The notation of the including element is depicted within the notation of the included one.

Figure 22: Architectural Model Relationship Notation

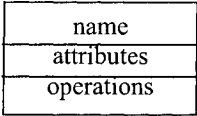
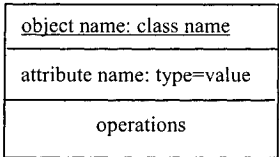
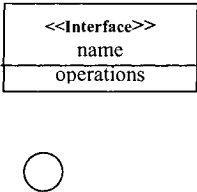


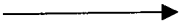

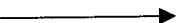
Element	Notation	Description
Class		It is drawn as a solid-outline rectangle with three compartments separated by horizontal lines, to hold the name, attributes, and operations of the class. The signature of an operation may be italicized to indicate the operation is abstract. An abstract class is depicted similar to a class, but with italicized class name.
Object		An object is depicted similarly to a class, but with instance-like characteristics. The top compartment shows the name of the object and its class with underlined format. The second compartment shows the attributes for the object and their values.
Interface	Or 	An interface may be shown with a full rectangle symbol with two compartments and the keyword <<interface>>. The name of the interface is put inside the upper compartment, and a list of operations supported by the interface is placed in the lower compartment. Sometimes an interface is depicted by a small circle for simplicity.
Template		A template class is depicted similarly to a class, but has a small dashed rectangle superimposed on the upper right-hand corner of the class rectangle. The dashed rectangle contains a parameter list of formal parameters for the class.
Collaboration		A collaboration is rendered as a dashed ellipse containing the name of the collaboration.
Message		A message is shown as a horizontal solid arrow from one instance to another instance. The arrow is labeled with the name of the operation to be invoked
State		A state is shown as a rectangle with rounded corners.
Transition		A transition is shown as a solid line originating from the source state and terminated by an arrow on the target state. A transition may be labeled by additional information.

Figure 23: Design Model Element Notation

Figure 23 and Figure 24. The case study is implemented with the C++ programming language.




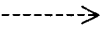
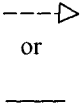
Relationship	Notation	Description
Generalization		A generalization is shown as a solid-line path from the child to the parent, with a large hollow triangle at the end of the path pointing to the parent.
Association		A solid line that connects to the two classes that has the association relationship. A name string can be used to indicate the meaning of the path. An association class is depicted as a class symbol which is attached to the association line with a dashed line. An association can be adorned by different kinds of optional property adornments, such as role names. A role is represented by a name string near the end of the association path.
Composition		A composition relationship between two classes indicates that one class is a part of the other one. The filled diamond is attached to the composite.
Dependency		The arrow begins with the depending element, and ends at the depended element
Realization		The line notation is chosen if the corresponding interface is depicted as a circle; otherwise the dashed line with a hollow arrow is used.
Multiplicity	m .. n	A multiplicity is a kind of properties of an association. It indicates the allowable range of the cardinality of the set of instances of the classes that the association connects. A multiplicity is represented by two integers as the lower bound and upper bound of the range. A star (*) can be used to represent unlimited non-negative integer range.

Figure 24: Design Model Relationship Notation

3.2 Alignment Maps between Models

Frameworks are more difficult to design and develop than individual applications. A framework has to not only capture commonalities of all applications that might be

built from the framework, but also possess enough flexibility to allow for the variations that exist between those applications [FHLS99]. Therefore, the requirement analysis of frameworks must perform both commonality and variability analysis. Existing Top-Down development approaches have suggested using domain analysis to capture framework requirements [STAR96] [FSJ99]. The domain knowledge is captured in range of outputs, including the context, taxonomy, data dictionary, feature model, domain specific software architecture, use cases and algorithms, and exemplar systems in the domain. Moreover, the commonality and variability should be preserved in the framework design as hot spots and frozen spots [WGM88] [SCHM97]. Therefore, it is desirable to have explicit guidelines that map the requirements to the design of frameworks.

Framework development requires an iterative approach in which the framework is refined a number of times [BOOC94] [JF88]. Bottom-Up approaches emphasize framework refactorings to evolve a framework to a mature and stable reusable platform [JOHN93] [RJ97]. From the traceability perspective, the changes should be propagated to the design and implementation of the framework. The consistency amongst the requirement, design, and source code must be identified and preserved. However, none of the existing methodology has addressed this issue [RJ97] [FSJ99] [SCHM97] [JGJ97] [WL99].

We want to find a way to capture the commonality and variability in each of the models, and preserve the traceability between the models. Thus, the identified commonality and variability are realized by mapping them from the feature model and use case model, to the architectural model, design model, and the source code. Here, we will discuss how to encompass the commonality and variability in each of the models, and define a set of alignment maps to maintain the traceability.

3.2.1 Modeling Commonality and Variability

Feature models capture the variability between different applications within a domain. The commonality is represented as mandatory features, while the variability is modeled with variable features, i.e. optional and alternative features [KCH+90] [KKL+98].

Use case models are used to elicit and describe the functional requirements of

frameworks. Jacobson and other people have already extended the OOSE methodology from simple applications to product families with Reuse Driven Software Engineering Business (RSEB) [JCJO92] [JGJ97] [DW98] [GFA98]. The commonality can be represented by the generalization and inclusion relationship, while the variability can be specified with the extension relationship [BRJ99].

Architectural models describe the high-level design abstraction of frameworks in terms of layer, subsystem, class, and interface. Usage of hot spots and frozen spots, design patterns, and abstract class hierarchy have been suggested to model the commonality and variability of frameworks [JOHN93] [PG94] [GOF94] [SCHM97].

Design models illustrate the detail design of frameworks with classes and objects. An object-oriented framework always has a core of abstract classes which defines the basic architecture of the framework. Some of those abstract classes can be specialized by subclasses so polymorphism provides the variability that is expected by different applications [TALI95]. For Black-Box frameworks, composition might be preferred to inheritance [JF88]. Parameterization is another means of incorporating variability into a design [JGJ97].

Object-oriented programming languages such as C++ provide ample mechanisms to realize commonality and variability in source code [STRO97]. Hook and template methods (see Section 2.8) are used to implement hot spots [PREE99]. Other than what we have mentioned in the design level, standard library, template, and pre-processor directives are also beneficial to this goal.

3.2.2 Maps

The common and variable aspects identified during the analysis should be reflected appropriately into framework design [SCHM97]. Our methodology uses a set of models to specify the requirements, design, and implementation of a framework. These models have different views on a framework at different levels of abstraction, and model the commonality and variability with various techniques.

Traceability is key to impact analysis during software maintenance and evolution [KELL90]. Traceability scheme are defined to specify the horizontal traceability links [PB90] and preserve the commonality and variability between those models.

We use trace maps to record the horizontal traceability links between models. In general, we trace model elements to model elements, and trace model associations to

model associations. The common associations such as *generalization*, *is_part_of*, and *dependency* are typically preserved by the trace maps.

Furthermore we attempt to *align* commonality and variability across the models, i.e. model elements that are common across the product line are mapped to each other, and model elements that are variable across the product line are mapped to each other.

We have chosen models in our methodology to encourage ease of describing trace maps. For example, the FORM decomposition of the feature model into capability, operating environment, domain technology, and implementation technique specifies the concerns for each category of feature and determine which model is the range of the trace map for that category (see Section 3.2.2.2).

It is important to note that we do not consider the “symmetry” of mappings, i.e. range items might have additional information other than the part that corresponds to the semantics of the domain items. This is not a problem at the current stage since the methodology only considers forward traceability. However, it will probably become an issue if the reverse maps are desired in the future.

3.2.2.1 Terminology

The set of models under discussion are the feature model M_f , use case model M_u , architectural model M_a , design model M_d , and source code M_i of a framework as defined by the set of the metamodels.

A *trace map* T from model M_1 to model M_2 records horizontal traceability between the elements and relationships of the models. That is, it records the *dependency* of elements and relationships of M_2 on elements and relationships of M_1 :

$$\boxed{\forall m \in M_1, T(m) \in M_2 \text{ is dependent on } m.}$$

An *alignment map* T for model M_1 to model M_2 is a trace map which preserves commonality and variability. That is: if $m \in M_1$ is *common* across the product line, then $T(m) \in M_2$ is *common* across the product line; if $m \in M_1$ is *variable* across the product line, then $T(m) \in M_2$ is *variable* across the product line.

In our methodology, we assume that the trace maps used are indeed alignment

maps, so we use the term trace map throughout.

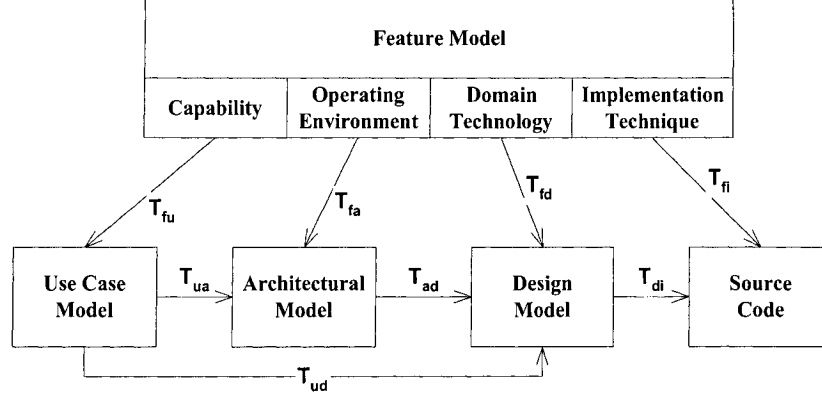


Figure 25: Trace Maps

The trace maps defined for the methodology (see Figure 25) are:

\mathbf{T}_{fu} : the trace map from the capability feature model to the use case model

\mathbf{T}_{fa} : the trace map from the operating environment feature model to the architectural model

\mathbf{T}_{fd} : the trace map from the domain technology feature model to the design model

\mathbf{T}_{fi} : the trace map from the implementation feature model to the source code

\mathbf{T}_{ua} : the trace map from the use case model to the architectural model

\mathbf{T}_{ud} : the trace map from the use case model to the design model

\mathbf{T}_{ad} : the trace map from the architectural model to the design model

\mathbf{T}_{di} : the trace map from the design model to the source code

For each trace map, we define the global constraint and concrete map rules for entities and relationships. The global constraint of a trace map is applied to all rules of that map by default, unless applicability issues are specified in the rule description. Ideally, a full map is desirable for each trace map. However, we have only defined partial maps at the current stage due to our limited experiences. Future work on trace maps is expected.

3.2.2.2 T_{fu}

Capability features of a framework characterize its services, functions, or non-functional constraints [KKL+98]. As a traditional problem in the literature of traceability [FINK91], it is difficult to map the features that represent those non-functional requirements to a use case model, other than as textual comments. We have not defined concrete map rules for the non-functional requirement features at the current stage.

Global Constraint : Any subset of capability features must be mapped to a (set of) service(s) described in the use case model, while preserving variability.

Rule 1 : Mandatory Feature \mapsto Use Case

A mandatory feature is a common property that must appear in all applications built from the framework. A use case describes common services that are provided by the framework. The map must preserve the variability.

Rule 2 : Optional Feature \mapsto Extending Use Case

The super-feature of the optional feature is mapped to the base use case of the extending use case. An optional feature may or may not exist in the applications built from the framework. The behaviour described by an extending use case is only performed when the extension point is reached. Issues about extension points are not considered in the rule.

Rule 3 : Alternative Feature \mapsto Extending Use Case

The super-feature of the alternative feature is mapped to the base use case of the extending use case. Alternative features represent different ways to configure their super-features. The variability is preserved by use case extension as same as Rule 2.

Rule 4 : Composed-of relationship \mapsto Inclusion relationship

If the sub-feature is a variable feature, the composed-of relationship is mapped to an extension relationship, and follows Rule 2 and Rule 3. Otherwise, it is mapped to an inclusion relationship. A composed-of relationship between capability features can be described as “a sub-service must participate in the execution of its super-service”. This can be specified with a use case inclusion

relationship. However, if the sub-feature is a variable feature, the variability has to be kept by an extension relationship. In a typical FORM model, the root capability feature often refers to the software being modelled. If a root feature represents the framework, composed-of relationships originated from the feature are not mapped.

Rule 5 : Generalization relationship \mapsto Generalization relationship

The parent feature is mapped to the parent use case, and the child feature is mapped to the child use case. The concept captured by a feature generalization relationship has nothing to do with variability, but specialization, which can be described by a use case generalization relationship.

Rule 6 : Implemented-by relationship \mapsto Inclusion relationship

Suppose feature f_1 is implemented by feature f_2 , then f_1 is mapped to the base use case; and f_2 is mapped to the included use case. The rationale behind the mapping is same as that of Rule 4. This rule is only applicable when both features are capability features. We only focus on horizontal traceability. An implemented-by relationship between features in different feature categories is vertical trace dependency, and has not been considered at the current stage. The limitation also exists in other feature categories.

3.2.2.3 T_{fa}

Operating Environment features represent attributes of the context in which a framework is employed. They describe the framework's conformance to the interfaces of external entities. At the current stage, we are only concerned about software related interfaces, and the architecture metamodel only consists of the logical architecture view. Therefore, hardware interface features are not mapped into the architectural model.

Global Constraint : Any subset of the operating environment features must be mapped to a (set of) interface(s) in the architectural model, while preserving variability. The protocol specified by the features must be abided by the corresponding interface(s). In terms of software interface, a protocol can be viewed as the service(s) requested by an external entity.

Rule 1 : Mandatory Feature \mapsto Interface

A mandatory feature in this category indicates an interface common to all applications built from the framework. An interface in the architectural model of a framework must be provided in all applications built from the framework. The variability is preserved.

Rule 2 : Optional Feature $\mapsto \emptyset$

A hot spot that supports the variability specified by the optional feature must exist in the architectural entity, which realizes the interface corresponding to the direct super-feature of the optional feature. Variability denoted by an optional feature cannot be directly realized by an interface alone. Instead, it is realized by a hot spot. Although an optional feature is not directed mapped into the architectural model, the mapping does not conflict with the general guidelines. As long as the “semantics” of two models correspond, their traceability is preserved even the structure of corresponding entities and relationships are different [LS96].

Rule 3 : Alternative Feature $\mapsto \emptyset$

A hot spot that supports the variability specified by the alternative feature must exist in the architectural entity, which realizes the interface corresponding to the direct super-feature of the alternative feature. The rationale behind the rule is same as that of Rule 2.

Rule 4 : Composed-of relationship \mapsto Inclusion relationship

If the sub-feature is a variable feature, the composed-of relationship is not mapped, because the variability has been provided via hot spots, as the description of Rule 2 and Rule 3. Otherwise, the super-feature is viewed as a set of distinct operations. The set is decomposed to a group of subsets. Each operation subset is represented by an interface, which corresponds to a sub-feature. Any operation inside the union of the subsets must exist in the interface that is mapped from the super-feature.

Rule 5 : Generalization relationship \mapsto Generalization relationship

The interface specified by the parent feature must be provided by the base class, and the interface specified by the child feature must be provided by the subclass.

Rule 6 : Implemented-by relationship \mapsto Dependency relationship

Suppose feature f_1 is implemented by feature f_2 , the interface specified by f_1 must be provided by the dependent architectural entity, and the interface specified by f_2 must be provided by the depended architectural entity. The rationale behind Rule 5 and Rule 6 are the general guidelines of relationship traceability mapping.

3.2.2.4 T_{fd}

Domain Technology features are a set of concepts, terminology, domain specific methods, and standardization, which are used for communication of stakeholders in a specific domain. We are only concerned about technical features. Other features such as business laws are not mapped into the design model. Objects derived from domain technology features encapsulate requirement decisions, and the object model is considered as an analysis model [KKL+98] [PR01]. So, those requirements should be fulfilled by the design decisions that are encapsulated by the classes in the design model. FORM encourages the usage of inheritance to model variability, but other mechanisms such as polymorphism and composition are also used in the map rules.

Global Constraint : Any subset of domain technology features is mapped to a class hierarchy in the design model. Requirements that are specified by the feature subset must be fulfilled by the design decisions which are encapsulated by the class hierarchy.

Rule 1 : Mandatory Feature \mapsto Class

If the feature has no direct variable sub-feature, it is mapped to a class. Otherwise, the feature is mapped to an abstract class that provides an interface to realize the required variability specified by the feature. A mandatory feature represents a common property of a product-line. There is no variability associated with the property, if the feature has no variable sub-feature. On the other hand, the variability indicated by the variable sub-features is realized by inheritance and polymorphism. Additional description is recommended to indicate that the abstract class represents a configurable common property.

Rule 2 : Optional Feature \mapsto Class

The optional feature is mapped to a subclass, and its super-feature is mapped to the abstract superclass of the subclass. The requirement is fulfilled by the class hierarchy. The variability is achieved with inheritance.

Rule 3 : Alternative Feature \mapsto Class

The alternative feature is mapped to a subclass, and its super-feature is mapped to the abstract superclass of the subclass. The requirement is fulfilled by the class hierarchy. Alternative features indicate different ways to configure the requirement that is captured by the super-feature. In the design model, the variability can be achieved by a set of concrete subclasses, which override the common interface defined by their superclass.

Rule 4 : Composed-of relationship \mapsto Composition relationship

If the sub-feature is a variable feature, the composed-of relationship is mapped to a generalization relationship, then follows Rule 2 or Rule 3. Variability specified by the variable feature is realized by inheritance. Otherwise, the super-feature is mapped to a composite class, and the sub-feature is mapped to a component class of the composite. It should be noted that if two classes have a composition relationship, the “composed” class is referred as the “composite” in the dissertation. It may be different to the Composite design pattern [GOF94].

Rule 5 : Generalization relationship \mapsto Generalization relationship

The parent feature is mapped to the superclass; and the child feature is mapped to the subclass.

Rule 6 : Implemented-by relationship \mapsto Dependency relationship

Suppose feature f_1 is implemented by feature f_2 , then f_1 is mapped to the dependent class; and f_2 is mapped to the depended class.

3.2.2.5 T_{fi}

Implementation Technique features characterize requirement decisions in terms of low-level implementation detail. They are more generic than the domain technology features and not limited in a specific domain. The availability of the realization of all implementation technique features is restricted by the development environment, and

the mechanisms provided by the source code languages. For a given domain entity or relationship, the mapping will be different in various implementation languages. Furthermore, Kelley [KELL90] states that the mapping from requirement to code is always many-to-many, which means that a requirement can be implemented by several code segments and a code segment can implement several requirements. This is a fundamental traceability problem, i.e. there is no natural one-to-one mapping from requirements to code. Thus, the map rules are not defined. However, it is sensible to assume that the semantics described by the implementation technique features can be realized by the C++ language with mechanisms such as inheritance, polymorphism, template, etc [STRO97].

Global Constraint : For any subset of implementation technique features, a finite number lines of code must exist in the source code to implement the requirements that are specified by the feature subset, without variability alteration.

3.2.2.6 T_{ua}

Use cases describe software services which are visible to external entities [JCJO92]. A framework use case model should also capture the commonality and variability encompassed by the framework. A framework architecture organizes the framework structure in terms of layers and subsystems, distributes the responsibilities to their interfaces, and realizes the required flexibility through hot spots and design patterns [DW98] [FSJ99]. As stated by Lindvall [LIND94], use case related traceability mapping are based on the functional requirements specified by use cases. Furthermore, mapping of commonality and variability should also be considered from the perspective of frameworks. Here, the trace map focuses on the mapping of services in terms of interfaces and operations in an architectural model. Variability encapsulated in a use case model with specific types of relationships such as inclusion and extension is also preserved by the mapping. At the current stage, services related to hardware entity are not mapped into the architecture model. Since a use case actor can be either a human being, or an external entity, actors are not mapped into the architectural model either.

Global Constraint : Any service described by use cases and episodes in the use case model must be provided by the layers, subsystems, and classes with their interfaces in the architectural model.

Rule 1 : Use Case \mapsto Interface

The service described by the use case is specified by the operation(s) in the interface, which is provided by a layer, a subsystem or a class. Here, the “use case” refers to a “generic” use case, which has no associated variability. The rationale behind the rule is that, use cases of a system describe the system’s visible behaviour, which are specified by interfaces in the architectural model of the system. Since the use case is a “generic” use case, no variability arises.

Rule 2 : Episode \mapsto Operation

The service described by the episode is provided by the operation. Here, we assume an episode as the atomic unit of use case models and does not embody any variability information. Wei [WEI04] introduces the concept of episode hierarchy in terms of episode tree to describe variability in a composite episode. Future work to accommodate the concept of episode tree into the mapping is expected.

Rule 3 : Extension relationship $\mapsto \emptyset$

A hot spot that provides the extensibility must exist in the architectural entity, which realizes the interface that is mapped from the base use case. Issues about extension points are not considered by the rule. The rationale behind this rule is similar to that of Rule 2 of T_{fa} (see Section 3.2.2.3).

Rule 4 : Generalization relationship \mapsto Generalization relationship

The services described by the parent use case must be specified by the interface that is provided by the superclass; and the services described by the child use case must be specified by the interface that is provided by the subclass. The operations involved in the two interfaces are inheritable. It is the typical application of traceable relationship mapping, i.e. generalization to generalization.

Rule 5 : Inclusion relationship \mapsto Dependency or Composition relationship

The included use case is mapped to an operation, and the base use case is mapped to an interface, if the variability specified by the inclusion can be achieved through the composition relationship between the operation and the interface. Otherwise, the base use case is mapped to an interface that is provided by the depending entity, which depends on the entity that provides the

operation mapped from the included use case. An inclusion relationship in a use case model means that the functionality described by the included use case is required to perform the services described by the base use case. It can be viewed as a dependency relationship. On the other hand, an included use case can be mapped to an operation that is regarded as a single operation interface, while the base use case is mapped to another interface which includes that operation. The rule is (1:many) type of traceability mapping (See 2.10.3).

3.2.2.7 T_{ud}

The design model of a framework specifies the detail design with classes and objects. The commonality and variability are realized with inheritance, composition, polymorphism, template, etc. Traceability mapping from use cases to classes and objects in design models is concerned with the functional requirements described by the use cases, and the participated objects involved in each of the use cases [LIND94]. We integrate this idea into the definition of the map rules. Every requirement specified by the use cases of the framework should be fulfilled by the design decisions that are encapsulated by the classes and their relationships. In addition, the commonality and variability must be preserved during the mapping. At the current stage, actors and hardware related use cases are not considered.

Global Constraint : Any service described by the use cases and episodes in a use case model must be provided by a class hierarchy in the design model, while preserving variability.

Rule 1 : Use Case \mapsto Interface

The service described by the use case is specified by the operation(s) in the interface, which is realized by a class hierarchy. Here, the class hierarchy consists of the classes of the participated objects of the use case. The rationale is similar to that of Rule 1 of T_{ua} . The “use case” refers to a “generic” use case, and no variability issue arises.

Rule 2 : Episode \mapsto Operation

The service described by the episode is provided by the operation. Actually, it is sensible to assume that the functionality described by an episode can always

be specified by an operation, although that operation might have to be fine-grained.

Rule 3 : Inclusion relationship \mapsto Dependency or Composition relationship

The included use case is mapped to an operation, and the base use case is mapped to an interface, if the variability specified by the inclusion can be achieved through the composition relationship between the operation and the interface. Otherwise, the base use case is mapped to an interface that is provided by the depending class, which depends on the class that provides the operation mapped from the included use case. The rationale is similar to that of Rule 5 of T_{ua} .

Rule 4 : Extension relationship \mapsto Abstract Class

The services described by the base use case are mapped to the virtual functions in the abstract class. The behaviour described by the extending use case is specified by the overridden operations in a subclass of the abstract class. An extension relationship implies that a use case might extend (“augment”) the behaviour described in the base use case, restricted by certain condition. The variability is achieved by overriding the polymorphic operations in the design model. Issues about extension points are not considered.

Rule 5 : Generalization relationship \mapsto Generalization relationship

The services described by the parent use case must be specified by the interface provided by the superclass; and the services described by the child use case must be specified by the interface provided by the subclass. The interfaces are composed of operations. The range subclass operations inherit those range superclass operations in order to keep the consistency of the semantics, i.e. generalization.

3.2.2.8 T_{ad}

Framework design consists of architectural design and detailed design [KARL95]. Framework architecture can be viewed as a set of design decisions within the framework and its smaller component [DW98]. The design model must preserve those

design decisions. The services provided by the layers and subsystems in the architectural model should be realized by the design model classes and their relationships. Hot spots and design patterns are often specified with class hierarchies in object-oriented application frameworks. Their transition into the design model should not be difficult, although some classes in the architectural model may need refinements.

Global Constraint : Any design decision in the architectural model must exist in the design model.

Rule 1 : Interface \mapsto Interface

The design model interface provides all services specified by the interface in the architectural model. Both architectural and design models use interfaces to provide functionality, which is specified by certain requirements. The functionality is regarded as the traceable information between the two models.

Rule 2 : Class \mapsto Class

The design model class encapsulates the design decisions which are provided by the class in the architectural model. The “class” concept in the range of T_{ad} covers abstract class, template class, and class hierarchy. This convention is applicable for the rest of the map rules.

Rule 3 : Layer \mapsto Class

The class provides the services that are provided by the layer. Variability embodied by the layer must be preserved by the map. A layer can be viewed as a facade class that provide its services through its interface [GOF94].

Rule 4 : Subsystem \mapsto Class

The class provides the services that are provided by the subsystem. Variability embodied by the subsystem must be preserved by the map. A subsystem can be viewed as a facade class that provide its services through its interface.

Rule 5 : Operation \mapsto Operation

The service that is specified by the operation in the domain must be specified by the range operation. If the domain operation is abstract, then the range operation must be abstract. The operation attributes must not be changed

by the map. Furthermore, if the domain operation is included in an interface provided by a layer or a subsystem, the range operation must be included in the interface of the class, which is mapped from that layer or subsystem.

Rule 6 : Composition relationship \mapsto Composition relationship

The architectural entities that are connected by the domain composition relationship must correspond to the design model entities that are connected by the range composition relationship. The direction of the composition relationship must be preserved.

Rule 7 : Dependency relationship \mapsto Dependency relationship

A layer dependency relationship is mapped to a dependency relationship between the classes. The depending layer is mapped to the depending class, while the depended layer is mapped to the depended class. The same idea is used for subsystem dependency relationship mapping.

Rule 8 : Association relationship \mapsto Association relationship

An association relationship in an architectural model only has two attributes, name and two association ends. Similar to the process stated in Rule 6, the entities connected by the range association relationship must correspond to the entities connected by the domain association relationship. Although a UML class association relationship also has attributes other than name and attribute end, the difference does not cause any problem since we do not consider the reverse map, as explained in 3.2.2.

Rule 9 : Inclusion relationship \mapsto Dependency relationship

The including entity is mapped to the depending class, and the included entity is mapped to the depended class. The mapping must not violate Rule 2 or Rule 3 or Rule 4 if one of them is applicable. An inclusion relationship indicates that a layer or a subsystem contains other layers, subsystems, or classes. The rationale of the rule is to view a layer or subsystem as a facade class, and the nesting entities act as service providers.

Rule 10 : Realization relationship \mapsto Realization relationship

The domain entity that realizes the domain interface is mapped to a class in the range, which realizes the range interface that corresponds to that domain interface.

Rule 11 : Generalization relationship \mapsto Generalization relationship

The architectural entities that are connected by the domain generalization relationship must correspond to the design model entities that are connected by the range generalization relationship, i.e. the parent class in the architectural model is mapped to the parent class in the design model, the child class to the child class.

3.2.2.9 T_{di}

The output of detail design are classes with attributes and methods which are specified with the target implementation language [KARL95]. The design decisions must be realized, and variability of the framework kept with mechanisms provided by the language. Other than the problem explained in 3.2.2.5, the map is also restricted by the target language. For example, implementation of an inheritance relationship in a language that supports multiple inheritance, may be different to implementation in a language that does not support multiple inheritance. Therefore, the map rules are not defined.

Global Constraint : Any design decision in the design model must be realized by a finite number lines of code in the implementation.

3.2.2.10 Summary

We have elaborated the set of trace maps amongst the models. The metamodels defined in the previous section are concerned with vertical traceability, which is specified within each of the models. Horizontal traceability is explicitly defined with the trace maps. The traceability web formed by the vertical and horizontal traceability fully conveys the traceable information of a framework. In next section, we will discuss the cascaded refactoring methodology, which utilizes the traceability links between the models to assist framework evolution.

3.3 Cascaded Refactoring Methodology

Industry experiences have proven that frameworks facilitate software reuse by providing a generic architecture with variable parts to reduce the cost of developing a family of applications in a given domain [TALI95] [DW98] [FSJ99]. However, no consensus has been made on a mature framework development methodology. We classified the existing methodologies into four categories: Bottom-Up [JF88] [JOHN93] [RJ97], Top-Down [STAR96] [CHW98] [WL99] [CN02], Hot Spot Generalization [PG94] [SCHM97] [PREE99], and Use Case Driven [JCJO92] [JGJ97] [DW98]. They are not mutually exclusive. For example, all of them suggest use hot spots and design patterns in the framework design to provide flexibility. The domain engineering process in Top-Down approaches analyze the exemplar applications to find the commonality and variability. The similar process is also utilized by Use Case Driven approaches to characterize reusable components. Furthermore, techniques from different categories can be combined, such as the integration of feature modeling into RSEB [GFA98]. Therefore, an intuitive idea of framework development is derived from them:

1. Perform an analysis on the target domain and specify the framework requirements in terms of commonality and variability with a feature model or a use case model.
2. Design the framework architecture with hot spots and design patterns to meet those requirements. The architecture design can also be aided by generalizing the architecture of existing applications in the domain, suggested by Bottom-Up approaches.
3. Refine the architecture to the detail design in terms of classes and objects. The commonality and variability are realized with usages of inheritance, composition, polymorphism, etc.
4. Implement the framework and test. Applications are also built from the framework to verify its flexibility and extensibility.
5. Evolve the framework until it reaches a mature platform.

The idea raises a couple of issues. First, the requirements captured during the analysis must be satisfied by the design, and realized in the implementation. The

commonality and variability of the framework must be preserved through its development. Second, framework development is an iterative process [FSJ99]. The evolution is different to that of individual applications in the “maturity” property. Other than enhancing the functionality as for individual application evolution, the maturity level of a framework also has to be improved from White-Box to Black-Box to Visual Builder [RJ97]. Furthermore, the impact of changes on the design and implementation due to the alteration on requirements has to be clearly illustrated and managed with traceability links. Third, the iteration and evolution demands strong support of framework documentation. Good documentation is also beneficial for application developers in framework customization [BD99].

The existing development methodologies have little work done on the three issues. We have refined the previous idea and proposed a moderate solution, called the cascaded refactoring methodology, to address those issues [BX01] [BCC+02] [BUTL02].

3.3.1 Cascade of Refactorings

The cascaded refactoring methodology is a hybrid approach which combines the modeling aspects of Top-Down domain engineering approaches, and the iterative refactoring process from the Bottom-Up approaches. Framework development is viewed as framework evolution, which is achieved by framework refactoring followed by framework extension. The methodology focuses on framework refactoring, and extends the notion of refactoring that has been applied to source code, and to design in the form of class diagrams, to other models of frameworks. It describes a framework with a set of models and relates the set of refactorings across those models through change impact analysis using the trace maps. The set of models are:

M_f : the feature model that organizes the common and variable features.

M_u : the use case model that captures the requirements.

M_a : the architectural model that describes the high-level design in terms of layers and subsystems

M_d : the design model that consists of static class hierarchy and dynamic object behaviour.

M_i : the source code implementation

The methodology stresses traceability between the models. In the context of a framework or product line, the methodology must also stress the distinction between commonality and variability. The trace maps preserve the traceability of common and variable features between the models. The set of trace maps is:

T_{fu} : the trace map from the capability feature model to the use case model

T_{fa} : the trace map from the operating environment feature model to the architectural model

T_{fd} : the trace map from the domain technology feature model to the design model

T_{fi} : the trace map from the implementation feature model to the source code

T_{ua} : the trace map from the use case model to the architectural model

T_{ud} : the trace map from the use case model to the design model

T_{ad} : the trace map from the architectural model to the design model

T_{di} : the trace map from the design model to the source code

The process of *cascaded refactoring* is a series of refactorings of the models, M_f to M_i . The impact of the refactorings on a model M_i to the refactorings on M_j , is translated via the trace maps that have M_i as the domain and M_j as the range. For example: Let f_1 be a mandatory sub-feature in the capability category. An include use case mapped from f_1 , named u_1 , must exist in the use case model, according to Rule 4 of T_{fu} . Assume refactorings on the feature model change f_1 to an optional sub-feature, then u_1 must be changed from an included use case to an extending use case, to maintain T_{fu} . Thus, refactorings are *cascaded* from the feature model to the use case model in order to preserve the traceability between the two models.

Refactorings of a framework are achieved through cascaded refactorings with one of the following paths with the format: “ $M_i \Rightarrow M_j$: T_{ij} ”. That is, refactorings of M_i determine the constraints on the refactorings of M_j via T_{ij} .

- **Capability Feature refactoring path:**

1 $M_f \Rightarrow M_u$: T_{fu}

$$2 \ M_u \Rightarrow M_a: T_{ua}$$

$$3 \ M_a \Rightarrow M_d: T_{ad}$$

$$4 \ M_d \Rightarrow M_i: T_{di}$$

- **Operating Environment feature refactoring path:**

$$1 \ M_f \Rightarrow M_a: T_{fa}$$

$$2 \ M_a \Rightarrow M_d: T_{ad}$$

$$3 \ M_d \Rightarrow M_i: T_{di}$$

- **Domain Technology feature refactoring path:**

$$1 \ M_f \Rightarrow M_d: T_{fd}$$

$$2 \ M_d \Rightarrow M_i: T_{di}$$

- **Implementation technique feature refactoring path:**

$$1 \ M_f \Rightarrow M_i: T_{fi}$$

The starting points of the four paths fully cover the feature model categories. Refactorings on the feature model can be cascaded step by step to the design and implementation of the framework. Trace maps support the refactoring process and deliver a solid foundation for framework evolution.

Given a source code program, a set of input values should result in the same output values before and after refactorings. This is regarded as the preserved “behaviour” of program refactoring [OPDY92]. Refactoring of design also treats functionality as the refactoring invariant. The notion of “behaviour” of the feature model, use case model, and architectural model is defined (see Section 3.3.2 for detail) to facilitate the description of refactorings on those models. The process of the cascaded refactorings of a framework is composed of two stages in terms of granularity:

Stage 1 : Refactorings at an individual model

Stage 2 : Cascaded refactorings between models

Any time during the framework refactoring is either at Stage 1 or at Stage 2. The refactoring rules of each model preserve the “behaviour” at Stage 1. Once refactorings on the domain of a trace map finish, the entity of the trace map will be updated according to the constraints prescribed by the map rules of that trace map. Thus, the trace maps keep the traceability when refactorings are being cascaded between models at Stage 2. The “behaviour” of a framework during refactorings is preserved from the requirements to the design and implementation via trace maps. The refactoring of a framework is *a set of model transformations that maps a coherent set of aligned models to another coherent set of aligned models*.

Refactoring of source code and design is to redistribute classes, variables, and methods across the class hierarchy for the ease of future adaptations and extensions [FOWL99]. In the context of a framework, refactorings should improve flexibility and extensibility. Especially when refactorings is the preliminary step to extensions, as the basic philosophy of the methodology. However, little work has been done on framework extension at the current stage due to our limited experience.

3.3.2 Model Refactoring

Refactoring of source code and design regard functionality as primary and to be preserved while other quality attributes such as performance take a secondary role. However, framework refactoring must not only consider functionality, but also flexibility in terms of common and variable aspects. Thus, we must define the appropriate notion of “behaviour” for the refactoring of the feature model, the use case model, and the architectural model.

3.3.2.1 Refactoring of Feature Model

A feature model does not only contain features of functionality, but also other properties such as non-functional requirements, even non-technical features such as business law. Furthermore, feature variability must be preserved during refactorings. Hence, it is not sensible to restrict the notion of “behaviour” to just features of functionality.

A feature model defines a collection of valid feature sets. In the context of a framework or a product line, an application is specified by a valid *feature set*, i.e. the set of all features provided by the application [KCH+90] [KKLL99]. If refactoring of source code of an application is to preserve the functionality of the application,

the refactoring effect on a valid feature set, which specifies an application built from the framework, must preserve every property described by the set of features and their relationships. So, refactoring of a framework feature model must preserve the collection of valid feature sets, which specifies all applications that can be built from the framework. We define the following rules for refactoring of feature model:

Rule 1 : A refactoring of a feature model M_f preserves the collection of M_f 's valid feature sets, each of which specifies an existing application that has been created from the product line. That is,

Assume APP_i be an existing application that has been built from the product line, which is specified by M_f ; $Fset_i$ be the feature set of APP_i ; S the collection of feature sets of the existing applications:

$\forall Fset_i \in S, Fset_i$ is *valid* in the post-refactoring model M'_f .

Rule 2 : To apply a feature model refactoring R to a feature model M_f , the precondition of R must be satisfied by M_f .

3.3.2.2 Refactoring of Use Case Model

Use cases capture the functionality of a system in a given environment. A use case can be divided into a cohesive set of episodes, each of which represents a sub-task [RAB96]. The functionality of a system can be defined as the set of episodes in the use cases of the system. In a context of framework, a use case model must also specify commonality with the generalization and inclusion relationship, variability with the extension relationship amongst use cases.

Refactoring of a framework use case model should preserve not only the functionality, but also commonality and variability. At the current stage, the methodology limits the notion of “behaviour” of use case model refactoring to only the “functionality”. Refactoring of use case model preserves the set of episodes. We define the following rules for refactoring of use case model:

Rule 1 : A refactoring of a use case model M_u preserves the set of episodes of M_u .

That is,

Assume e_i be an episode of the pre-refactoring M_u ; S the set of episodes of M_u :

$\forall e_i \in S, e_i$ exists in the post-refactoring model M'_u .

Rule 2 : To apply a use case model refactoring \mathbf{R} to a use case model M_u , the precondition of \mathbf{R} must be satisfied by M_u .

3.3.2.3 Refactoring of Architectural Model

Software architecture provides an abstract description of the organisational and structural decisions that are evident in a software system. There are too many quality attributes relevant to architecture to decide which properties should be preserved by transformations, to the best of our knowledge. It is possible to have a set of quality-preserving refactorings for each quality attribute. This situation might become even more complex in the context of a framework, since architecture of frameworks also contain hot spots and frozen spots to realize variability and commonality. At the current stage, the methodology takes the default quality attribute, i.e. functionality.

Typically, a use case view can be taken for the subsystems in architecture [BCKR97]. Services of a subsystem are accessed through the interfaces of the subsystem. Functionality of a system can be viewed as a set of services that are provided by the subsystems in the system architecture. In the context of our architectural metamodel, refactoring of architecture should preserve the set of services in terms of operations, which are provided by the architecture. We define the following rules for refactoring of architectural model:

Rule 1 : A refactoring of an architectural model M_a preserves the set of services of M_a . That is,

Assume s_i be a service of the pre-refactoring M_a ; S the set of services of M_a :

$\forall s_i \in S, s_i$ is provided by the post-refactoring model M'_a .

Rule 2 : To apply an architectural model refactoring \mathbf{R} to a architectural model M_a , the precondition of \mathbf{R} must be satisfied by M_a .

3.3.3 Documenting Refactoring

A design rationale is “a representation of the reasoning behind the design of an artefact” [KEAN97]. It refers to why a certain design decision is made and which requirements is realized the decision [CONK89]. A rationale records the assumptions,

arguments, and decisions behind a particular design, to allow reviewers and maintainers follow the previous reasoning used by the designers. The iterative process of framework evolution expects consistent and comprehensive documentation, as typical software application do. The rationale behind refactorings of a framework should be recorded to assist the evolution. The methodology views a refactoring as an issue-driven activity. Since framework refactoring is achieved through cascaded refactorings of a set of models of the framework, the overall rationale of refactoring is a collection of *decisions*. Each decision records a refactoring performed on one model of the framework. The process of cascaded refactoring is documented by a sequence of refactorings, each of which is described by a decision record. Thus, the refactoring document presents a clear *roadmap* of the sequence of refactorings on the involved models, in order to evolve the framework.

<p>Decision Record 2:</p> <p>Intent: Make the ConfigureImp a subclass of the Configure abstract class.</p> <p>Choice: Inherit</p> <p>Arguments: ConfigureImp class, Configure class</p> <p>Validation: The ConfigureImp class becomes a subclass of the Configure class. Since the ConfigureImp class has neither parent class nor subclass before the refactoring, as stated in Decision Record 1, the precondition of the refactoring is satisfied. The Configure's virtual function in the Configure class must be overridden by the ConfigureImp class to provide the default Configure service,</p>

Figure 26: A Decision Record Example

It is also important to capture the rationale behind the individual model refactoring. So, the purpose of the refactoring, the choice of refactorings, and the reasons for its choice should be recorded. We suggest that the decision records the following information:

intent: the motivation and purpose of this step of restructuring

choice: refactorings that are appropriate for the change, or refactorings for a high-level refactoring that is composed of a series of lower-level refactorings

arguments: the parameters for the choice, with a possible discussion of trade-off analysis for other candidate refactorings

validation: the consequences of this step of restructuring, with a possible discussion of how to preserve behaviour with the preconditions

Of course, this rationale should relate to the assumptions and priorities documented in the original design rationale. Figure 26 gives a decision record example.

3.3.4 Refactorings

The definition of a refactoring often includes invariants (“preserved behaviour”) that should remain satisfied and pre and postconditions that should hold before and after the refactoring has been applied. Opdyke suggests the use of preconditions to ensure the behaviour preservation [OPDY92]. Roberts extended his definition of refactoring by adding postconditions, which are assertions that a program must satisfy for the refactoring to be applied [ROBE99]. Heckel has formally proved that any set of refactoring postconditions can be translated into an equivalent set of preconditions [HECK95]. The methodology defines a refactoring in the following format:

Name: a meaningful name to specify the intent of the refactoring

Description: a textual explanation of the refactoring activity

Parameters: the model entities and relationships involved in the refactoring

Preconditions: the context that must be satisfied to execute the refactoring

A set of refactorings is defined for the feature model, the use case model, and the architectural model, respectively. These lists of refactorings are not complete. However, they are sufficient for our case study. More refactorings will be added in the future.

3.3.4.1 Feature Model Refactorings

We use S to represent the collection of feature sets of the existing applications in all feature model refactorings.

The first set of refactorings modify the variability of a feature:

Name: `Change_mandatory_to_optional`

Description: takes a mandatory feature f and makes it optional

Parameters: f ;

Preconditions: $\forall \text{Fset}_i \in S, f \notin \text{Fset}_i$.

Name: Change_optional_to_mandatory

Description: takes an optional feature f and makes it mandatory

Parameters: f ;

Preconditions: $\forall \text{Fset}_i \in S, f \in \text{Fset}_i$.

Name: Change_optional_to_alternative

Description: takes an optional feature f at a variation point V_p and makes it alternative

Parameters: $f; V_p$;

Preconditions: $\forall \text{Fset}_i \in S$, if $f \in \text{Fset}_i$, f takes and only takes one of the enumerated values f_1, f_2, \dots, f_n that are represented by the alternative features.

Name: Change_alternative_to_optional

Description: takes a set of alternative features f_1, f_2, \dots, f_n at a variation point V_p and makes an optional feature f

Parameters: $f; f_1, f_2, \dots, f_n; V_p$;

Preconditions: $\exists \text{Fset}_i \in S, \exists f_{t(1 \leq t \leq n)} \in \text{Fset}_i$. f_t is optional at V_p .

Name: Add_alternative

Description: takes a set of alternative features f_1, f_2, \dots, f_n at a variation point V_p and adds another feature f_{n+1} to the list of alternatives

Parameters: $f_1, f_2, \dots, f_n, f_{n+1}; V_p$;

Preconditions: True.

The second set of refactorings restructures the composed-of hierarchy:

Name: Add_optional

Description: takes a feature f_s and makes an optional feature f to be the sub-feature of f_s

Parameters: $f_s; f$;

Preconditions: $\forall \text{Fset}_i \in S$, if $(f_s \in \text{Fset}_i) \wedge (f \in \text{Fset}_i)$, f is a sub-feature of f_s .

Name: Promote_feature

Description: takes a feature f that is a sub-feature of f_s and promotes it in the hierarchy to be a sibling of f_s

Parameters: $f_s; f;$

Preconditions: $\forall \text{Fset}_i \in S, (f_s \notin \text{Fset}_i) \vee (f \notin \text{Fset}_i)$

Let us choose **Add_alternative** as an example to illustrate the behaviour preservation of feature model refactorings. Assume all existing applications that have been created so far from the product line have different values at a variation point V_p , each of those values is represented as feature $f_{t(1 \leq t \leq n)}$. After the refactoring, any feature set derived from the post-refactoring feature model can take $f_{t(1 \leq t \leq n+1)}$ at V_p . Any existing feature set that is prior to the refactoring still conforms to the post-refactoring feature model no matter which f_t is chosen at V_p , because $f_{t(1 \leq t \leq n)}$ is a subset of $f_{t(1 \leq t \leq n+1)}$. Thus, the precondition is always true. That is: $\forall \text{Fset}_i \in S, \text{Fset}_i$ is *valid* in the post-refactoring model M'_f . The collection of valid feature sets is preserved.

The precondition of **Change_alternative_to_optional** is derived from the classification in Generative Programming [CE00], that is, any alternative feature set can be normalized to contain either alternative optional features, or alternative features (each of them is not optional).

3.3.4.2 Use Case Model Refactorings

We use U_{name} to represent the set of existing use case names, A_{name} as the set of existing actor names, and $\text{EP}(u)$ as the set of episodes included in a use case u , in all use case refactorings.

The first set of refactorings are concern about restructuring of actors and behaviour:

Name: **Create_abstract_actor**

Description: identifies two actors a_1 and a_2 and create a common super-actor a as their parent

Parameters: $a_1; a_2; a;$

Preconditions: assume u_1 as the use case that can be associated with a_1 ; u_2 as the use case that can be associated with a_2 ; u as the use case that can be associated with a . $\text{EP}(u_1) = \text{EP}(u_2)$, or u_1 and u_2 have an extension relationship with u . $a \notin A_{\text{name}}$.

Name: **Create_abstract_usecase**

Description: identifies two use cases u_1 and u_2 as specializations of a common super use case u

Parameters: $u_1; u_2; u;$

Preconditions: $(EP(u_1)=EP(u_2)) \wedge (u \notin U_{name}).$

Name: Merge_actors

Description: identifies two actors a_1 and a_2 as a common actor a

Parameters: $a_1; a_2; a;$

Preconditions: $a \notin A_{name}.$

Name: Split_actor

Description: identifies the special cases a_1 and a_2 of an actor a

Parameters: $a_1; a_2; a;$

Preconditions: assume $EP(u_{s(1 \leq s \leq m)})$ as the set of episodes included in the use cases associated with a_1 ; $EP(u_{t(1 \leq t \leq n)})$ as the set of episodes included in the use cases associated with a_2 ; $EP(u_{w(1 \leq w \leq k)})$ as the set of episodes included in the use cases associated with a . $\forall e_i \in EP(u_{w(1 \leq w \leq k)}) \Rightarrow (e_i \in EP(u_{s(1 \leq s \leq m)})) \vee (e_i \in EP(u_{t(1 \leq t \leq n)}))$. $(a_1 \notin A_{name}) \wedge (a_2 \notin A_{name}).$

Name: Merge_behaviours

Description: identifies two use cases u_1 and u_2 as a common use case u

Parameters: $u_1; u_2; u;$

Preconditions: $\forall e_i \in (EP(u_1) \cup EP(u_2)) \Rightarrow e_i \in EP(u). u \notin U_{name}.$

The next set of refactorings redistributes behaviour in the form of episodes from one use case to another. These are similar to the refactorings of a class hierarchy that move methods:

Name: Make_episode_usecase

Description: takes a use case u with an episode e and creates a new use case u_1 with behaviour precisely described by e . A relationship link u includes u_1 is added

Parameters: $u; e; u_1;$

Preconditions: $\forall e_i \in EP(u), e_i \neq e \Rightarrow e_i \notin EP(u_1). e \in EP(u). u_1 \notin U_{name}.$

The last set refactorings restructure use case generalization hierarchy:

Name: Move_episode_to_parent_usecase

Description: takes a use case u with child use cases $u_{i(1 \leq i \leq n)}$, each of which includes a common episode e , and moves e to u

Parameters: $u; e; u_{i(1 \leq i \leq n)}$;

Preconditions: $e \notin EP(u). \forall i \in (1 \leq i \leq n) \Rightarrow e \in EP(u_i)$.

Name: `Move_episode_to_child_usecase`

Description: takes a use case u with an episode e , moves e to each of the child use cases of u

Parameters: $u; e; u_{i(1 \leq i \leq n)}$, as the child use case of u ;

Preconditions: $e \in EP(u). \forall i \in (1 \leq i \leq n) \Rightarrow e \notin EP(u_i)$.

Let us choose **Move_episode_to_child_usecase** as an example to illustrate the behaviour preservation of use case model refactorings. In terms of episodes, the change only relates to the episodes included in u and episodes included in the child use cases of u . Prior to the refactoring, episode e is not included in any child use case of u , as dictated by the precondition. However, e can be inherited by all the child use cases from u , according to the definition of use case generalization relationship. After the refactoring, e is distributed into every child use case and removed from u . The distribution will not cause any conflicts to the existing episodes of any child use case, in order to satisfy the precondition. The set of episodes of u and its child use cases are not changed before and after the refactoring. Thus, the behaviour is preserved.

3.3.4.3 Architectural Model Refactorings

The work on architectural refactorings focuses on the changes related to the interfaces of layers or subsystems of architecture. We use SUB_{name} to represent the set of existing subsystem names, LAY_{name} as the set of existing layer names, INT_{name} as the set of existing interface names, $Operation(K)$ as the set of operations included in an interface K , $Interface(K)$ as the set of interfaces provided by the subsystem (layer) K , and $Client(K)$ as the set of clients that are depended on the interface K in all architectural refactorings.

The first set of refactorings manage the creation or name change of interfaces, subsystems, and layers:

Name: `Create_subsystem`

Description: creates a new subsystem S

Parameters: S ;

Preconditions: $S \notin \text{SUB}_{\text{name}}$.

Name: Create_layer

Description: creates a new layer L

Parameters: L ;

Preconditions: $L \notin \text{LAY}_{\text{name}}$.

Name: Change_interface_name

Description: takes an interface I and changes its name to I_1

Parameters: $I; I_1$;

Preconditions: $I_1 \notin \text{INT}_{\text{name}}$. $\forall C_i \in \text{Client}(I) \Rightarrow C_i \in \text{Client}(I_1)$. $\forall \text{Op}_i \in \text{Operation}(I) \Rightarrow \text{Op}_i \in \text{Operation}(I_1)$.

Name: Change_subsystem_name

Description: takes an subsystem S and changes its name to S_1

Parameters: $S; S_1$;

Preconditions: $S_1 \notin \text{SUB}_{\text{name}}$. $\forall C_i \in \text{Client}(\text{Interface}(S)) \Rightarrow C_i \in \text{Client}(\text{Interface}(S_1))$.

Name: Change_layer_name

Description: takes a layer L and changes its name to L_1

Parameters: $L; L_1$;

Preconditions: $L_1 \notin \text{LAY}_{\text{name}}$. $\forall C_i \in \text{Client}(\text{Interface}(L)) \Rightarrow C_i \in \text{Client}(\text{Interface}(L_1))$.

The second set of refactorings looks at the interfaces of a subsystem or a layer and re-distributes their operations. Here, although the description is based on subsystem, all refactorings are also applicable for layers:

Name: Split_interface

Description: takes an interface I of a subsystem S and redistributes the operations of I across two new interfaces I_1 and I_2 of the subsystem S

Parameters: $I; I_1; I_2; S$;

Preconditions: $(I_1 \notin \text{INT}_{\text{name}}) \wedge (I_2 \notin \text{INT}_{\text{name}})$. $I \in \text{Interface}(S) \Rightarrow (I_1 \in \text{Interface}(S)) \wedge$

$(I_2 \in \text{Interface}(S)). \forall \text{Op}_i \in \text{Operation}(I) \Rightarrow (\text{Op}_i \in \text{Operation}(I_1)) \vee (\text{Op}_i \in \text{Operation}(I_2)).$

Name: Merge_interfaces

Description: takes two interfaces I_1 and I_2 of a subsystem S and combines the operations of I_1 and I_2 together to a new interfaces I of the subsystem S

Parameters: $I; I_1; I_2; S;$

Preconditions: $I \notin \text{INT}_{\text{name}}. (I_1 \in \text{Interface}(S)) \wedge (I_2 \in \text{Interface}(S)) \Rightarrow I \in \text{Interface}(S). \forall \text{Op}_i \in (\text{Operation}(I_1)) \cup (\text{Operation}(I_2)) \Rightarrow \text{Op}_i \in \text{Operation}(I).$

The third set of refactorings redistribute the services provided by a subsystem or a layer. Usually these are accompanied by a redistribution of interfaces and structure. According to the architectural metamodel, a service is specified by operations. Thus, we assume that a service can always be specified by an interface which includes all operations that are required to provide the service. We use “ \neq ” to specify the “not equal to” relationship between two different interfaces. In the following refactorings, a service is regarded as an interface, in order to simplify the description of the refactorings:

Name: Move_service_to_sibling

Description: takes a service s of subsystem S_1 with sibling subsystem S_2 in a hierarchical client-supplier architecture and assigns it to S_2

Parameters: $S_1; S_2; s;$

Preconditions: $(s \notin \text{Interface}(S_2)) \wedge (s \in \text{Interface}(S_1)). (\forall I_t \in \text{Interface}(S_1)) \wedge (I_t \neq s) \Rightarrow \text{Operation}(I_t) \subseteq (\text{Operation}(\text{Interface}(S_1)) \setminus \text{Operation}(s)). (\forall c \in \text{Client}(s)) \wedge (s \in \text{Interface}(S_1)) \Rightarrow (s \in \text{Interface}(S_2)) \wedge (c \in \text{Client}(s)).$

Name: Delegate_service_to_supplier

Description: takes a service s of subsystem S_1 with a supplier subsystem S_2 and assigns the service to S_2

Parameters: $S_1; S_2; s;$

Preconditions: $(s \notin \text{Interface}(S_2)) \wedge (s \in \text{Interface}(S_1)). (\forall I_t \in \text{Interface}(S_1)) \wedge (I_t \neq s) \Rightarrow \text{Operation}(I_t) \subseteq (\text{Operation}(\text{Interface}(S_1)) \setminus \text{Operation}(s)).$

Name: Promote_service_from_internal

Description: takes a service s provided by a nested subsystem S_2 of a subsystem S_1 and makes it a service of S_1

Parameters: $S_1; S_2; s;$

Preconditions: assume $\text{InterClient}(s)$ as the set of S_1 's nested subsystems which are clients of s ; $(s \notin \text{Interface}(S_1)) \wedge (s \in \text{Interface}(S_2))$. $(\forall I_t \in \text{Interface}(S_2)) \wedge (I_t \neq s) \Rightarrow \text{Operation}(I_t) \subseteq (\text{Operation}(\text{Interface}(S_2)) \setminus \text{Operation}(s))$. $\text{InterClient}(s) \subseteq \{S_2\}$. $(\forall c \in \text{Client}(s)) \wedge (s \in \text{Interface}(S_2)) \Rightarrow (s \in \text{Interface}(S_1)) \wedge (c \in (\text{Client}(s) \cup \{S_2\}))$.

Name: Demote_service_to_internal

Description: takes a service s provided by a subsystem S_1 with a nested subsystem S_2 and makes s a service of S_2

Parameters: $S_1; S_2; s;$

Preconditions: $(s \notin \text{Interface}(S_2)) \wedge (s \in \text{Interface}(S_1))$. $(\forall I_t \in \text{Interface}(S_1)) \wedge (I_t \neq s) \Rightarrow \text{Operation}(I_t) \subseteq (\text{Operation}(\text{Interface}(S_1)) \setminus \text{Operation}(s))$. $(\forall c \in \text{Client}(s)) \wedge (s \in \text{Interface}(S_1)) \Rightarrow (s \in \text{Interface}(S_2)) \wedge (c \in \text{Client}(s))$.

It should be noted that the last precondition of **Promote_service_from_internal** is different to that of **Move_service_to_sibling**. The reason is explained with the following example. Assume there are two subsystems in architecture, S_1 and its nested subsystem S_2 . The service s is provided by S_2 and its clients are S_1 and S_2 (Self dependency). If s is promoted from S_2 to S_1 , then S_2 should not be a client of s because S_2 cannot request services from its outer subsystem S_1 , according to the architectural metamodel. Thus, S_2 should be added into the set of clients of s , to ensure the correctness of the last precondition in **Promote_service_from_internal**. The same rationale is behind the precondition $\text{InterClient}(s) \subseteq \{S_2\}$, which means that other than S_2 itself, no nesting subsystems of S_1 should be the clients of s .

Move_service_to_sibling and **Delegate_service_to_supplier** are applicable to layers. However, issues arise when the other two refactorings are applied directly on layers, because the different constraints between layers and subsystems, i.e. any entities inside a layer can only interact with other external entities through the layer interface. The inclusion relationship of layers is merely a client-supplier structure. Thus, **Demote_service_to_internal** is same as **Delegate_service_to_supplier**, in the context of layers. Here, we give the description of **Promote_service_from_internal**

for layers:

Name: `Promote_service_from_internal`

Description: takes a service s provided by a nested layer L_2 of a layer L_1 and makes it a service of L_1

Parameters: $L_1; L_2; s;$

Preconditions: assume $\text{InterClient}(s)$ as the set of L_1 's nested subsystems which are clients of s ; $(s \notin \text{Interface}(L_1)) \wedge (s \in \text{Interface}(L_2))$. $(\forall I_t \in \text{Interface}(L_2)) \wedge (I_t \neq s) \Rightarrow \text{Operation}(I_t) \subseteq (\text{Operation}(\text{Interface}(L_2)) \setminus \text{Operation}(s))$. $(\forall c \in \text{Client}(s)) \wedge (s \in \text{Interface}(L_2)) \Rightarrow (s \in \text{Interface}(L_1)) \wedge (c \in (\text{Client}(s) \cup \text{InterClient}(s)))$.

Let us choose **Move_service_to_sibling** as an example to illustrate the behaviour preservation of architectural model refactoring. According to the architectural metamodel, a service is specified as operations. The set of services provided by the pre-refactoring model can be divided into three parts: the services provided by S_1 , services provided by S_2 , and services provided by the rest of the architectural elements other than S_1 and S_2 . For the third part, only the clients that request s are effected. In the post-refactoring model, S_1 's services other than s are still available as specified by the second precondition. S_2 's services is "augmented" by s because s does not collide with the interfaces of S_2 prior to the refactoring, according to the first precondition. Thus, the union of S_1 's services and S_2 's services are as same as that in the pre-refactoring model. The last statement of the precondition assures that the change has been propagated to all clients of s , and their services will not be effected. Thus, the total set of services of the pre-refactoring model is preserved by the refactoring.

3.3.5 Conclusion

Refactoring of source code has long been used for Bottom-Up development and evolution of object-oriented frameworks. The concept of refactoring is extended to the feature model, the use case model, and the architectural model, of an object-oriented framework. Refactoring of these models is the first step in framework evolution: refactoring and extension.

In the cascaded refactoring methodology, a framework is described by a set of models: a feature model organizing common and variable features; a use case model

of requirements; an architectural design; a design showing class collaborations; and source code. The overall refactoring of the framework is a set of refactorings of the models, and the constraints on how to refactor a particular model is determined or impacted by the previous refactorings. Hence, the restructuring cascades from one model to the next. The cascading of impacts of changes follows the trace maps between models. Overall, a refactoring of a framework is a set of model transformations that maps a coherent set of aligned models to another coherent set of aligned models. The methodology is unique in three ways:

1. To view refactoring as an issue-driven activity
2. To document the rationale of an application of a refactoring as a triple: intent of restructuring, choice of refactoring(s), and impact of the restructuring
3. The notion of cascaded refactoring, where the restructuring of one model determines constraints on the restructuring of other models (via the trace maps)

The three issues introduced at the beginning of this chapter have been addressed by the methodology. Commonality and variability of a framework are identified and captured into a feature model. The set of trace maps keep the horizontal traceability and flexibility links from the feature model to the design model and source code. So, the realization of required variability is achieved via aligning the models.

Framework evolution is regarded as framework refactoring followed by framework extension. The methodology focuses on framework refactoring at the current stage. Framework refactoring is achieved by applying a set of refactorings on the models of a framework. In addition, requirement alteration is appropriately propagated to the design and code with the trace maps. The precise definition of models, refactoring rules, and refactoring document are beneficial to framework documentation.

While the working set of partial models is incomplete, and hence some mappings for traceability or alignment are partial maps, a consistent, coherent, aligned set of models and maps is desired. Our work is ongoing, particularly in terms of enumerating all the refactorings of the particular models, and in investigating whether there is a need for architectural refactorings that preserve quality attributes other than functionality.

Chapter 4

Know-It-All Case Study

Be true to your work,
your word, and your friend.
~Henry David Thoreau

A case study is developed to validate the cascaded refactoring methodology. The case study is a framework for relational Database Management System (DBMS), called **Know-It-All**. The DBMS domain is chosen because:

1. The DBMS domain is mature with many available resources for feature oriented domain analysis, such as open source DBMS applications, published papers, books, and domain experts, etc.
2. The DBMS domain is stable without lots of rapid changes. It is a good candidate for building frameworks [TALI95].
3. The need for DBMS applications, as well as their use, is still rapidly growing [RG00].

Our aim is not to provide a powerful relational DBMS with complex structures for that will demand much more resources than we can afford. **Know-It-All** has only a small subset of DBMS features as follows:

- Create and modify a relational database with a given schema
- Verify and load a relational database from files into the main memory

- Query data with a simple data manipulation language

Features such as transaction management and data updates are not considered.

The remainder of this chapter is organized as follows. Section 4.1 introduces the case study. The Know-It-All models are presented in section 4.2. The trace maps amongst the models are illustrated in section 4.3. Section 4.4 demonstrates the cascaded refactoring methodology with two refactoring examples. Section 4.5 discusses what has been learnt from the case study.

4.1 Case Study

The research on software methodology in an academic setting needs a concrete case study for the purpose of validation. Our case study, Know-It-All, is a RDBMS framework to validate the cascaded refactoring methodology.

4.1.1 Introduction to the Domain

A *database* is a collection of data, which describes the activities of one or more related organizations [RG00]. A *database management system* (DBMS) is a software system designed to manage and utilize large collections of data. The objective of using a DBMS is to provide a convenient and effective method of defining, storing, and retrieving the data contained in its database. A *data model* is a collection of high-level data description constructs that hide many low-level storage details. A DBMS allows its users to define the stored data in terms of data models. The relational data model is the dominant data model in DBMS community, and has been used in many commercial database systems, such as Informix, Oracle, Sybase, and Microsoft SQL Server [RG00]. There are also other existing data models, such as the hierarchical model, object model, object-relational model, and network model. A *relational database* is a database in which the data is logically perceived as tables, which are concrete instances of relations. A *relational database management system* (RDBMS) manages tables of data and associated index structures that increase functionality and performance of tables. A database *schema* is a description of data in terms of a data model. The schema of a relation describes its name, the name of each field (*attribute*), and the data *type* of each field. A *data definition language* (DDL) is used to define the schema of a database.

The data in a DBMS can be viewed as three levels of abstraction: *external*, *conceptual*, and *physical*. The database description consists of a schema at each of them. The conceptual schema specifies the stored data in terms of the data model of the DBMS. The external schema also describes the data in terms of the data model, and allows data access to be customized for individual users or user groups. A *view* is conceptually a relation, but the records in a view are not stored in the DBMS; instead, they are computed using a definition for the view in terms of relations stored in the DBMS. An external schema is composed of a collection of views and relations from the conceptual schema. The physical schema describes additional storage detail, such as how to store the relations on secondary storage devices. *Indexes* are auxiliary data structures created to speed up data retrieval operations. The schema information is stored in the *system catalogs*.

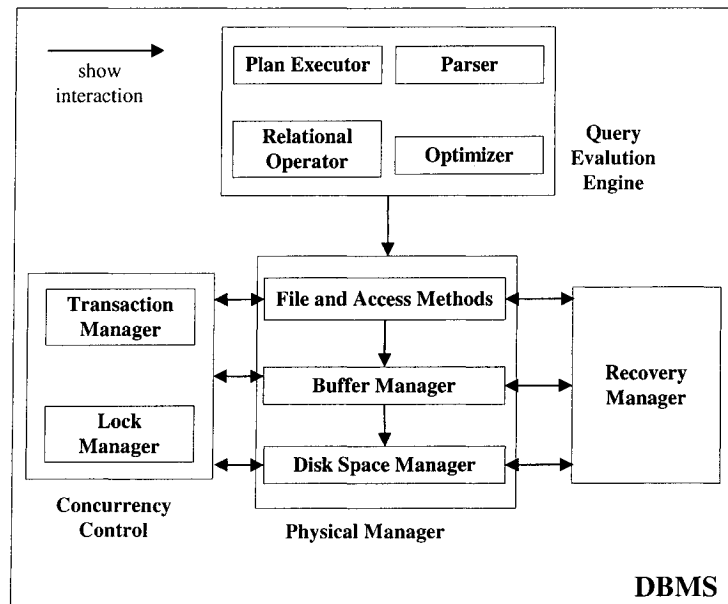


Figure 27: A Simplified Relational DBMS Architecture

The usage of a DBMS does not only include data storage, but also data retrieval. A *query* is a question involving the data stored in a DBMS. A *data manipulation language* (DML) is used to create, modify, and query data in a DBMS. Query processing is one of the most important features of a DBMS. The other features include transaction management, data integrity, and crash recovery.

A simplified relational DBMS structure (Figure 27 [RG00]) can be divided into

three parts: Query Evaluation Engine, Concurrency Control, and Physical Manager. When a DBMS user issues a query, the Parser translates the query to be a parsed query and sends the parsed query to the Optimizer that uses the information of the data storage to generate efficient execution plans for the query evaluation. An *execution plan* usually consists of relational algebra expressions. The execution plan is sent to the Plan Executor subsequently to get the query result.

A *file* in a DBMS is a collection of pages or a collection of records. The implementation of relational operators depends on the File and Access Methods layer. The layer includes a variety of software to support files and indexes. The Buffer Manager brings pages from secondary storage devices to main memory as needed in response to read requests. It relies on the Disk Space Manager, which hides details of the underlying operating system and hardware, and allows higher levels of the DBMS to view the data as a collection of pages.

The DBMS supports concurrency and crash recovery by carefully scheduling user requests and maintaining a log of all changes to the database. A *transaction* is any one execution of a user program in a DBMS. The Transaction Manager ensures that transactions request and release locks according to a suitable locking protocol and schedules the transaction execution. A *lock* is a mechanism used to control access to database objects. The Lock Manager keeps track of requests for locks and grants locks on database objects when they become available. The Recovery Manager is responsible for maintaining a log, and restoring the system to a consistent state after a crash.

There are several groups of people associated with the creation and use of databases. DBMS software is built by *database implementers*. The *end users* store and use data in a DBMS. They often just simply use software applications that are made by *Database Application Developers* to interact with the DBMS. Those applications facilitate the DBMS usage by hiding the technical knowledge required to use the DBMS. Enterprise-wide databases are typically important and complex enough to be managed and maintained by professionals, called *Database Administrators* (DBA). They are responsible for the design of conceptual and physical schemas, security and authorization, data availability and failure recovery, and database tuning.

4.1.2 Introduction to the Framework

The aim of the Know-It-All framework is to support a variety of data models of data and knowledge, different paradigms of integration, and heterogeneous databases [BCC+02]. It will be used to customize advanced database applications in bioinformatics. Know-It-All is designed with scientific databases in mind, and does not provide transactions. Instead, it provides a data feed mechanism for bulk or incremental data loads. The prime concern is querying the existing data. The framework provides a generic infrastructure for DBMS and supports a range of data models (relational, object, object-relational, etc) where the data model itself, and its constituents for query language, query optimizing, indexing, and storage have clearly defined roles.

A database in Know-It-All is seen as a series of layers, each of which provides the same interface. The usual breakdown of responsibilities into physical, logical, conceptual, and view layers is followed by Know-It-All. Each layer in Know-It-All is basically a translator between its client layer and its supplier layer. A layer provides a mechanism to decompose or translate queries, and a mechanism to reconstruct answers (for example, an execution plan for relational algebra expressions). The translation is done with the aid of the schema, and produces both the translated query, and the mechanism to reconstruct answers. The layer architecture is adapted from one for heterogeneous databases [MB96], while the reconstruction is done by an iterator tree to get the result. Know-It-All will eventually incorporate composite databases (such as integrated or heterogeneous databases) and make no distinction between simple and composite databases.

Limited by resources, to date Know-It-All only implements relational DBMS, because the relational data model is mature, and many applications are available as sources for the bottom-up refactoring work [BBG+89] [RAMA96] [MB96]. The first version prototype has been implemented with GNU C++, with some Java for user interface, and XML for communication of data between the C++ framework and the Java tools. It supports query processing, data feed and schema definition. It also contains two sub-frameworks: OPT++ for query optimization [KD99] and Gist for index techniques [HKP97]. The prototype provides a generic infrastructure for relational database management systems and components for query optimizing, indexing, and storage management. ANSI SQL is chosen as the query language, for its popularity in relational database domain. Flat text files are used as the storage medium.

4.2 Case Study Models

A framework is specified by its feature model, use case model, architectural model, design model, and source code. Refactoring of a framework is achieved through a series of cascaded refactorings performed on its models. The traceability of those models is kept by the trace maps among them. This section presents the **Know-It-All** models except its source code; the trace maps will be illustrated in the following section.

4.2.1 Feature Model

The features (Figure 28) are collected from various sources in the DBMS domain such as textbooks, papers, exemplar systems, and expert feedbacks [BBG+89] [DESA90] [RAMA96] [MB96] [BP98] [RG00] [KWON03]. The features of **Know-It-All** are organized into four categories: Capability, Operating Environment, Domain Technology, and Implementation Technique.

In the Capability category, the **Know-It-All** framework supports mandatory features **Query**, **Administration**, and **DataFeed**. The **Query** feature represents the mechanism for retrieving information from a database with questions in a predefined format. It has two mandatory sub-features: **DBConnect**, connecting to the specific database to issue the query, and **DBDisconnect**, close the connection to the queried database. The **Administration** feature stands for the management services such as schema definition and database tuning. It is composed of five sub-features. The mandatory features **DBInitialize** and **DefineSchema** represent the function of database creation and database schema definition. The **Configure** feature denotes the service of database environment customization with given parameters. The **Tune** feature indicates the service of modifying a database to ensure adequate performance as user requirements change. The **Monitor** feature specifies the service for a DBA to supervise the DBMS related activities. The **Configure**, **Tune**, and **Monitor** feature are optional features. The **DataFeed** feature specifies the service of loading data into the DBMS. It has two child features: **IncrementalUpdate** and **BulkLoad**, which specify two different ways of feeding data into the DBMS. The non-functional requirements **Performance** and **Scale** are optional features. The **Performance** feature represents the DBMS efficiency and the **Scale** feature indicates the availability of the DBMS to handle greater

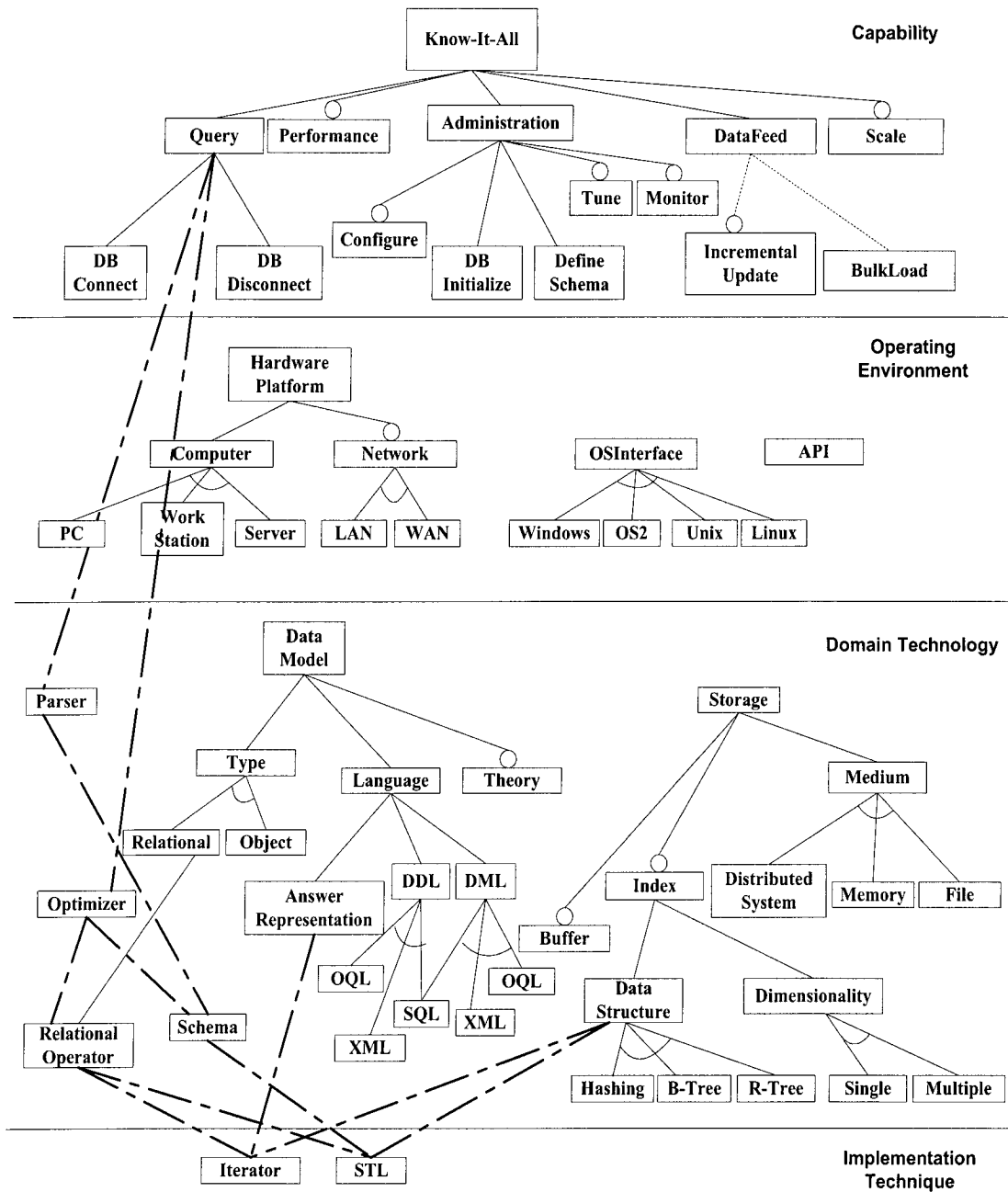


Figure 28: Know-It-All FORM Feature Model

usage demands.

The Operating Environment features represent the environmental constraints and both hardware and software interfaces of Know-It-All. The **HardwarePlatform** feature represents the hardware environment with which Know-It-All interacts. It consists of the mandatory **Computer** feature, and optional feature **Network**, which can be configured to either LAN (Local Area Network) or WAN (Wide Area Network). The **Computer** feature and its alternative sub-features indicate the different computer environments in which Know-It-All can operate: a standalone PC, a workstation or a server in a client-server setup. The **API** feature stands for Application Programming Interface, which lets other software applications interact with Know-It-All. The **OSInterface** feature specifies the interface to operating systems. Its alternative sub-features represent different operating systems: Windows, OS2, Unix, and Linux.

The Domain Technology features of Know-It-All include the **DataModel**, **Storage**, **Optimizer**, **Parser**, and **Schema** feature. The **DataModel** feature represents the collection of high-level data descriptions supported by Know-It-All. It is composed of **Type**, **Language**, and **Theory** features. The **Type** feature and its alternative sub-features indicate two different kinds of data models, relational data model and object data model. The **RelationalOperator** feature represents the abstraction of relational algebra expressions. It supports the implementation of the **Optimizer** feature. The **Language** feature indicates the language used for data manipulation (DML), data definition (DDL), and the answer representation. The **Query** feature in the Capability category is implemented by the collaboration of the **Optimizer**, **Parser**, and consequently the **Schema** feature. There are many data definition languages (DDL) and data manipulation languages (DML). However, SQL is chosen in the prototype as both of them since SQL is the standard query language in the relational DBMS domain. The **OQL** (Object Query Language) and **XML** (eXtensible Markup Language) features indicate other query languages. The **AnswerRepresentation** feature specifies the format of the query result. The **Theory** sub-feature represents the theory supported by the data model, such as relational algebra. The **Parser** feature represents the component of translating queries with the assistance of the **Schema**, which stores the DBMS meta-information. The **Schema** is essential for most DBMS services. The **Optimizer** feature stands for the component that evaluates the best execution plan. The process is called “optimization”, which also needs the collaboration of the **Schema**. The **Storage**

feature characterizes physical storage issues. The **Medium** feature indicates the place where the data is stored, either in the main memory, or in files, or in distributed systems. The **Index** feature represents the indexing mechanism, which is important to improve the efficiency and performance of DBMS applications. Two indexing issues are considered: **DataStructure** and **Dimensionality**. The **DataStructure** feature and its alternative sub-features represent different index structures: Hashing, B-Tree (Balanced Tree), and R-Tree (Rectangle Tree). The **Dimensionality** feature and its alternative sub-features specify the dimensionality of the index data structure, which can be either single dimension, or multiple dimensions.

The Implementation Technique features are more generic than the Domain Technology features. For instance, the **Iterator** feature can also exist in other domains. In **Know-It-All**, the implementation of answer representation and relational operator relies on the iterator. The C++ Standard Template Library (STL) is heavily employed in the prototype. They are the two most important features for **Know-It-All** in this category. Other implementation features are not shown in the model.

There are three types of feature relationships in the model: Composed-of, Generalization, and Implemented-by. For instance, the **Configure** feature has a Composed-of relationship with the **Administration** feature; the relationship between the **BulkLoad** and **DataFeed** is Generalization; and the **Query** feature has an Implemented-by relationship with the **Parser** feature.

It should be noted that a typical FORM model also includes feature composition rules and feature selection decision records. Since we are mainly concerned with alignment mapping and refactorings, they are not considered in order to simplify our situation.

The features that have been implemented by the **Know-It-All** prototype are given in the following list; the rest of them are left for future work.

- Capability: **Query** and its sub-features, **Administration** and its mandatory sub-features, **DataFeed** and its mandatory child feature, **Performance**, and **Scale**.
- Operating Environment: **HardwarePlatform**, **Computer**, **PC**, **OSInterface**, **Unix**, and **API**.
- Domain Technology: **DataModel**, **Type**, **Relational**, **Language**, **AnswerRepresentation**, **DDL**, **DML**, **SQL**, **Parser**, **Optimizer**, **RelationalOperator**, **Schema**, **Storage**,

Medium, and File. To date the prototype has not incorporated the Gist index framework.

- Implementation Technique: Iterator and STL.

4.2.2 Use Case Model

The Know-It-All use case model categorizes the users to be DBA, Advanced User, and End User, shown in Figure 29. The DBA actor represents database administrators who perform management services. The **Administration** use case has two included use cases:

1. **InitializeDatabase**: a database is created with a given schema
2. **DefineSchema**: a DBA can define the schema for a database

In addition, the **Administration** use case can be extended by three extending use cases:

1. **Configure**: a DBA can customize the environment of a database with given parameters, such as how to backup files of a database
2. **Tune**: a DBA can modify the database to meet the expected performance, such as modifying the physical schema of a database to improve the search efficiency
3. **Monitor**: a DBA can examine the activities of a database

The **Advanced User** is responsible for feeding data to the databases in **Know-It-All**. The **DataFeed** use case can be specialized to **IncrementalUpdate** and **BulkLoad**, which represents two ways of data loading.

The **End User** actor represents the common “naive” users who search and use data in **Know-It-All**. The **Query** use case specifies the typical query-processing job performed by them. It has two included use cases:

1. **ConnectDatabase**: a specific database is connected and made ready for operations
2. **DisconnectDatabase**: a specific database is disconnected and not available for further activities

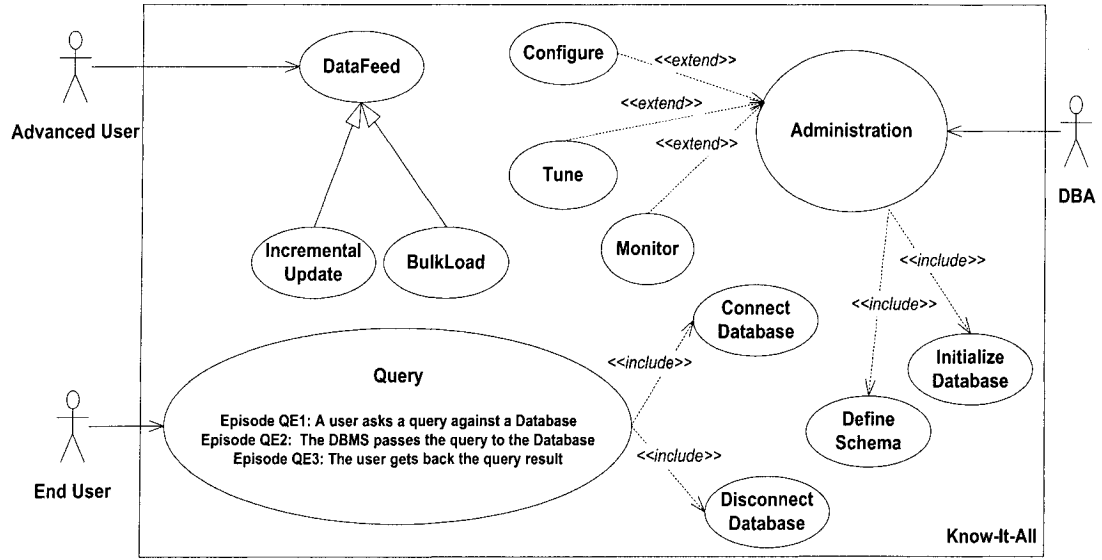


Figure 29: Know-It-All Use Case Model

The Query use case is also decomposed into further detail. There are three inherent episodes:

1. QE1: A user issues a query against a specific database in the DBMS
2. QE2: The query is sent to the database and processed
3. QE3: Once the process is done, the user can get the query result from the DBMS

4.2.3 Architectural Model

The Know-It-All architectural model is shown in Figure 30. The DBMS subsystem is responsible for database administration, data loading, and database access. It provides the interface LAPI and LKIAQuery for different kinds of users. The LAPI interface specifies the administration services. The operations included in the LAPI correspond to the included and extending use case of the Administration use case, respectively. For instance, the Initialize_DB operation can be used to create databases. Three abstract classes Configure, Monitor, and Tune provide the corresponding services in the LAPI interface. The LKIAQuery interface specifies the query service. The AskQuery operation is used to issue queries against a database in Know-It-All, and the query result can be returned one by one with the GetNext operation. The

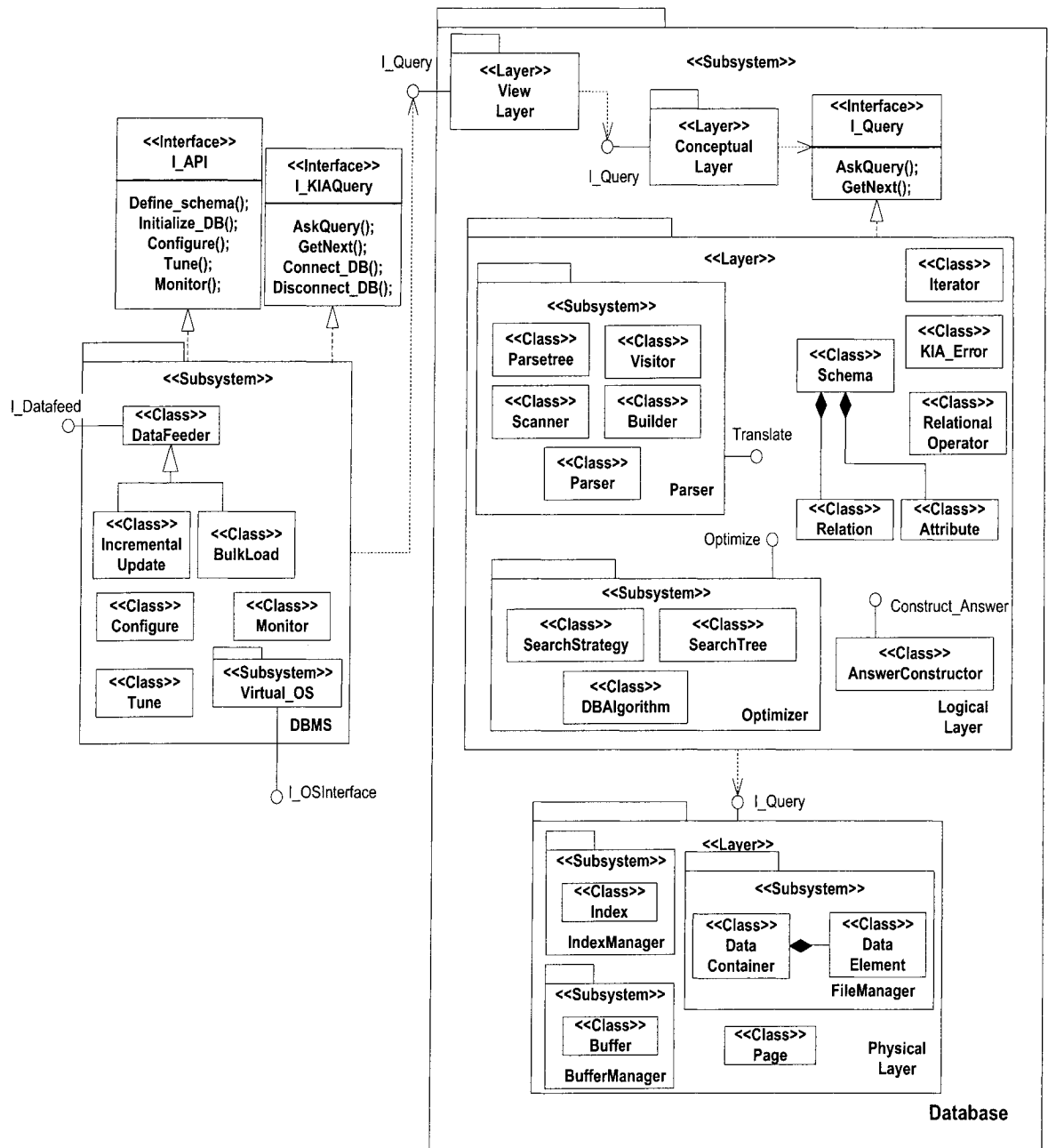


Figure 30: Know-It-All Architectural Model

Connect_DB and Disconnect_DB are used to open and close the connection to a specific database in order to perform queries.

The **DataFeeder** class hierarchy provides data feeding services with the **IDatafeed** interface. A hot spot is provided to support different feeding mechanisms: incremental load and bulk load. The variability can be achieved through class inheritance. The hot spot contains the base class (**DataFeeder**), which defines a common interface **IDatafeed**; and concrete derived classes (**IncrementalUpdate** and **BulkLoad**), each represents one of the different alternatives for the variable aspect. The **DataFeed** operation (shown in Figure 34) is a hook method. When the **DataFeed** operation in an object of the **DataFeeder** base class is called, the overridden **DataFeed** operation in an object of the derived class will be executed through a polymorphic reference typed with the base class. This is called “binding” the hot spot [SCHM97]. The **Virtual_OS** subsystem is responsible for the interaction with operating systems on which **Know-It-All** works. It provides the interface **ILOSInterface**, which includes the operations to interact with operating systems. The subsystem possesses a hot spot to cope with different operating systems. A base class with a set of hook methods which provide the common operations to deal with operating systems, and concrete derived classes which override those hook methods in order to be compatible with different operating systems.

The **Database** subsystem is decomposed to be a series of layers: **ViewLayer**, **ConceptualLayer**, **LogicalLayer** and **PhysicalLayer**. There are dependency relationships between these layers. Each of them is basically a translator between its client layer and its supplier layer. A layer provides a mechanism to decompose or translate queries, and a mechanism to reconstruct the answers. The translation is done with the aid of the schema to produce both the translated query, and the information to reconstruct the answers. Typically, a user is authorized to a view of a database. When he asks a query against a database, the query is defined in terms of his view of the database. The query is translated into a query in terms of real relations and attributes in the **ViewLayer**, and passed to the **ConceptualLayer** by calling the **AskQuery** method in the **IQuery** interface of the **ConceptualLayer**. Our initial design to have the **ConceptualLayer** is to deal with heterogeneous databases. In the **ConceptualLayer**, the query is decomposed to be sub-queries which target to various databases. The query results of the sub-queries will be reconstructed later to be the result of the original query.

To date we have not implemented the **ViewLayer** and **ConceptualLayer** in **Know-It-All**, and there are only stub classes in those layers. The **LogicalLayer** is used to translate the query into logical relational operators, and the **PhysicalLayer** addresses storage related issues. They are presented in the architectural model in detail.

There are two subsystems in the **LogicalLayer**. The **Parser** subsystem is responsible for translating a given query into a query tree and a reconstruction plan, if there is no error within the query. The subsystem provides the **Translate** interface, which can be called by its client. The class **Parsetree** is the intermediate representation of parsed queries. The parsing job is performed by the collaboration of **Scanner**, **Builder**, and **Visitor**. The **Scanner** class is responsible for interpreting a query into tokens, which is used by the **Parser** class. The **Builder** class is responsible for the creation of objects of the **Parsetree** class. The **Builder** and **Visitor** class adopts the **Builder** and **Visitor** design pattern, respectively [GOF94]. The **Optimizer** subsystem has a hot spot to support different optimization strategies. We integrate the **OPT++** optimizer framework into **Know-It-All** to provide the flexibility [KD99]. There are three main classes in the **Optimizer**. The **SearchStrategy** class defines the common interface for all search strategies that are used in query optimization. The **SearchTree** represents the search tree that is used to explore the optimal plan. The **DBAlgorithm** class defines the interfaces for all possible execution algorithms for the logical operators.

The **AnswerConstructor** class reconstructs query results according to reconstruction plans. Since we have not implemented the mechanism to support heterogeneous databases, there is no “actual” reconstruction from separate databases, simply (re)constructed results from the tables using the execution plan. In the current prototype, the plan is a tree of iterators. The **Iterator** class provides overloaded operators **++** and ***** to fetch the next query result. The **Schema** class represents the data model abstraction and stores meta-information of the database. Since **Know-It-All** is a RDBMS framework, the **Schema** class mainly pertains to relations and attributes, which are specified by the **Relation** and **Attribute** class. The **Schema** class allows its clients to obtain meta-information for query processing and answer reconstruction. It can be designed as a hot spot to support various data models for future extension. The **KIA_Error** class provides exception handling services.

The **PhysicalLayer** includes three subsystems: **BufferManager**, **IndexManager** and **FileManager**. The **BufferManager** subsystem manages the buffer used for data exchange

between the main memory and disk storage. The **Buffer** class is the abstraction of the data buffer. To date we have not implemented the mechanism, and leave it for future work. The **IndexManager** subsystem provides the indexing service to improve the efficiency of database access. The **FileManager** subsystem is responsible for the file access. The **DataElement** class is the abstraction of the data stored in a **Know-It-All** database. A **DataContainer** object is a container of **DataElement** objects. The container can be navigated by its iterator. Once a **DataContainer** object is created, an iterator is bound to it, and any further activity associated to the container is delegated to its iterator. The **Page** class represents the physical storage unit.

All four layers provide the common interface, **I.Query**, which allows its client to ask queries and obtain the corresponding result. The interface has two important operations: **AskQuery** and **GetNext**. The **AskQuery** operation returns an iterator, which is a root of the iterator tree corresponding to the result of the query. The **GetNext** operation sends back the result by calling the **operator++** on the returned iterator.

Know-It-All is a White-Box level framework. The customization is done by providing subclasses to the abstract classes in the framework. For some of the abstract classes, we have designed the concrete subclasses and the application developers only need to choose from them to meet their own specific purposes. The execution of **Know-It-All** is a single thread process running on a standalone PC with a single processor. The process model and deployment model are trivial and not integrated into the architectural model.

4.2.4 Design Model

The high level design is shown in Figure 31. To date we have only implemented the **LogicalLayer** and **PhysicalLayer** in the **Database** subsystem. The **DatabaseFacade** classes from the **ViewLayer** and **ConceptualLayer** are two stub classes, which provide the common interface **I.Query**. They are merely used to maintain the four-layer architecture in the **Database** subsystem. Thus, the class diagrams of those two layers are not given. The **DatabaseFacade** class in the **LogicalLayer** and **PhysicalLayer** are described in the following paragraphs. All classes which have a name ending with “Facade” are the application of the Facade design pattern. They are the clients of the abstract class **KIA.Error**, which provides the exception handling services.

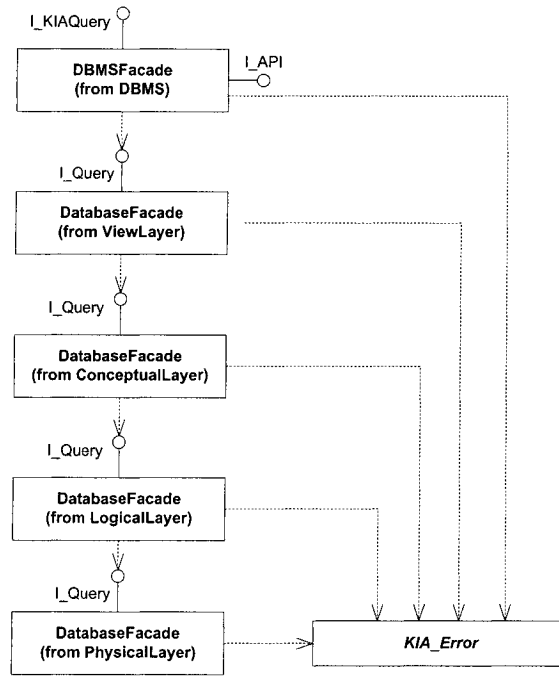


Figure 31: Know-It-All Design Model: High Level View

Figure 32 shows the class diagram of the **LogicalLayer** in the Database subsystem. The **DatabaseFacade** class represents the key concept, the database, and acts as a mediator that distributes responsibilities to corresponding components within the layer. It supports the interface **I_Query**, which is depended on the **DatabaseFacade** class located at the **ConceptualLayer**. Two operations are included in the **I_Query**: **AskQuery** operation takes a query as input and the result can be returned one by one by the **GetNext** operation. The **DatabaseFacade** has Factory methods to create objects of the **Parser**, the **OptimizerFacade**, and the **AnswerConstructor** class. The factory method is an application of the Abstract Factory design pattern, which is suitable for creating a family of related product objects. During query processing, the **DatabaseFacade** uses the **Parser** to translate the given query from the **DatabaseFacade** in the **ConceptualLayer**, and passes the parsed query to the **OptimizerFacade** to get the optimized execution plan. The query result is reconstructed by the **AnswerConstructor**.

The abstract class **DataModel** is composed of the **DataModelType** class and the **Language** class. The **Theory** class inherits the **DataModel** class to accommodate future extension, such as relational calculus support. A **DataModelType** class can be specialized to either a **RelationalDataModel** class or an **ObjectDataModel** class. The

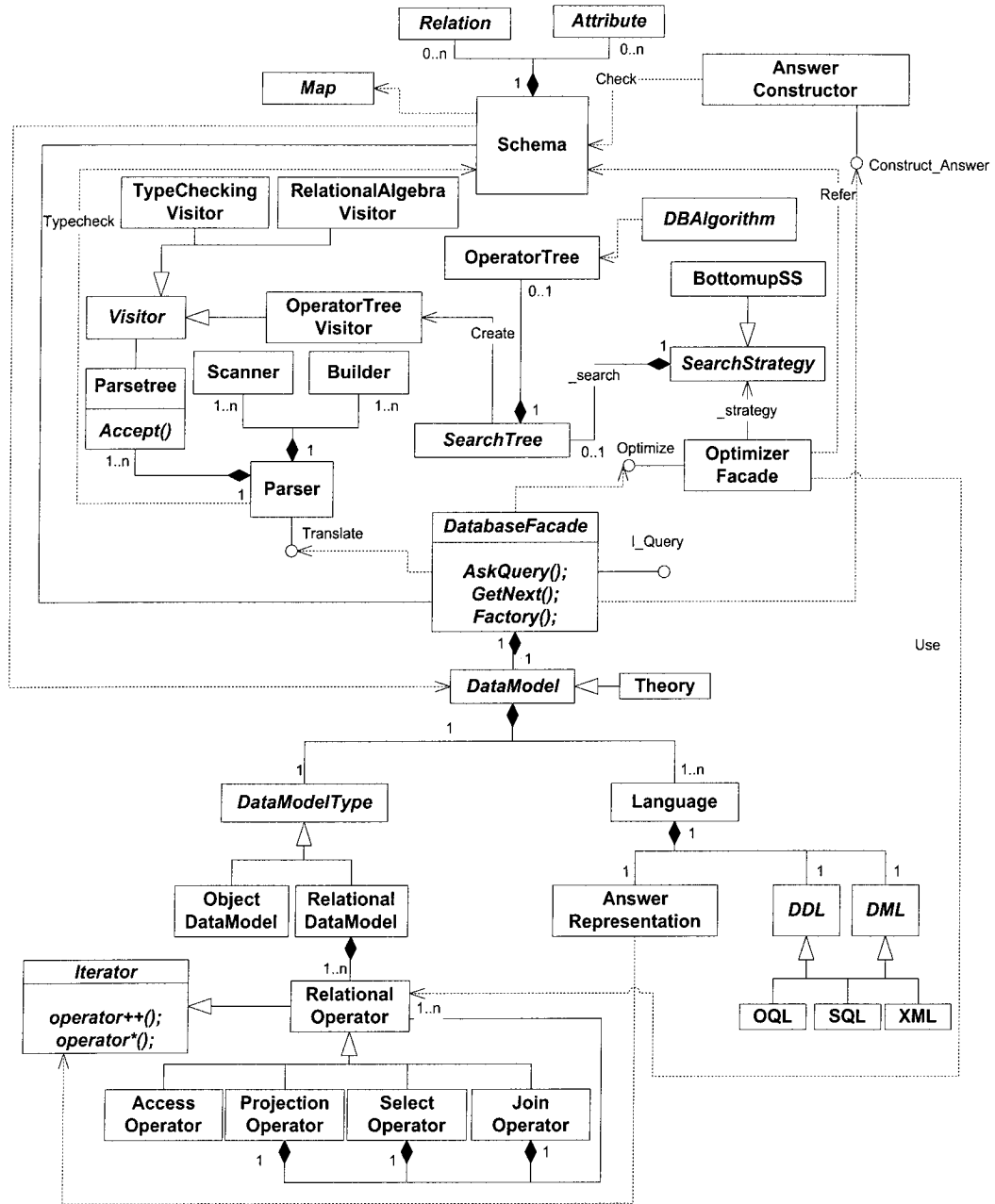


Figure 32: Know-It-All Design Model I: Logical Layer

former consists of `RelationalOperators`. The `Language` class encapsulates the language related information: the language for answer representation, for data definition, and for data manipulation. The `SQL` class inherits both `DDL` class and `DML` class because the `SQL` language possesses both `DDL` and `DML` features.

The `Parser` class is composed of the `Scanner` class and the `Builder` class. A `Scanner` object scans a query to produce a series of token. The `Parser` class relies on the `Builder` class to build a `Parsetree` with the tokens. The `Parsetree` class is the abstraction of the intermediate query representation. Each node of a `Parsetree` object associates to two specific kind of `Visitor` objects. The `TypeCheckingVisitor` class is for type checking with the information from the `Schema` class; and the `RelationalAlgebraVisitor` is used to create a relational algebra tree out of the `Parsetree` object. Once a `Parsetree` is produced, the two visiting processes are performed. Here, the `Builder` and `Visitor` design patterns are applied.

The `OptimizerFacade` class takes a relational algebra tree as the input and produces the optimal plan. It supports the `Optimize` interface. Different search strategies can be chosen through the initialization of corresponding `SearchStrategy` subclasses. During the query processing, an instance of a concrete subclass of the `SearchStrategy` is created by the `OptimizerFacade` object. The default search strategy provided by `Know-It-All` is the `BottomUp` strategy. However, advanced framework developers can change the search strategy or mix it with other strategies. The `SearchStrategy` class maintains a reference to the `SearchTree` class, which is also an abstract class. The `SearchTree` class represents the search space that is used to explore the optimal plan. The `OperatorTree` class is the abstraction of a logical query plan. It is an algebraic expression that represents the particular operations to be performed on data in specific orders. The `OperatorTreeVisitor` class is also a subclass of the `Visitor` class. The `DBAlgorithm` abstract class represents the physical algebra. It can be specialized to provide concrete algorithms to create physical nodes.

The logical plan produced by the optimization is a tree of `RelationalOperator` objects. A reference to the `RelationalOperator` class is kept by the `OptimizerFacade` class. The `RelationalOperator` class inherits from the `Iterator` abstract class, which is an application of the `Iterator` design pattern. The `RelationalOperator` class provides the common interface for its subclasses: `AccessOperator`, `JoinOperator`, `PredicateOperator`, and `SelectOperator`. The plan tree is a `Composite` design pattern.

The **AnswerConstructor** class reconstructs the query results passed by the **DatabaseFacade** class. Although the answer construction in the current prototype is performed by the execution plan, the extensibility is available for future accommodation of heterogeneous databases.

The **Schema** class is another key concept in the design. In the traditional database design, a catalog class is used to store all database-wise information regarding the relations, attributes, indexes and supplemental data such as statistics or cost. It is easy to design but difficult to evolve. **Know-It-All** uses a **Schema** class hierarchy to store the descriptive information of the databases. For each layer, there is a **Schema** subclass that only maintains the data of the specific layer. Since **Know-It-All** focuses on RDBMS at the current stage, the **Schema** consists of information of relations and attributes, which are represented by the abstract class **Relation** and **Attribute**, respectively. The **Map** class is a C++ standard template from the STL. It is used to build up the mapping information of relations and attributes for the **Schema** subclasses in different layers. In the **LogicalLayer**, the **Schema** class is used by other components during the query processing, such as the **Parser** class and the **OptimizerFacade** class.

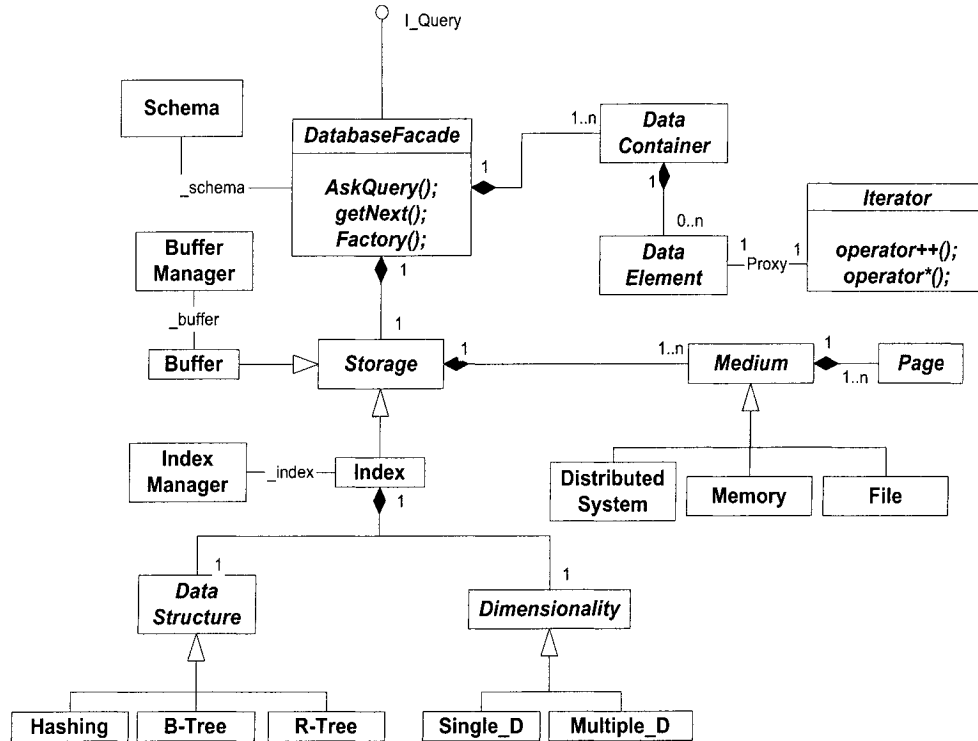


Figure 33: Know-It-All Design Model II: Physical Layer

The important property of database management systems is to persistently store data. In the **PhysicalLayer** (Figure 33), the **Storage** abstract class encapsulates the data store related information and services. It has a class composition relationship to the **Medium** class. The **Medium** can be specialized to be either a **DistributedSystem** class, or a **Memory** class, or a **File** class. **Know-It-All** use plain files as the default data store medium. The **DataElement** class is the abstraction of the data stored in a **Know-It-All** database. A **DataContainer** object is a container of **DataElement** objects. The container can be navigated by its iterator. It is also the application of the Iterator design pattern. The **Page** class represents the physical storage unit. The **DataElement** and **DataContainer** classes hide the actual physical data information, and maintain a clear mapping with the relational operators in the **LogicalLayer**.

The **Storage** class can be specialized to the **Buffer** and **Index** subclasses. The **BufferManager** class is responsible for the data exchange between the main memory and the secondary storage. There is no buffer mechanism available in the prototype.

Two properties of database indexes are dimensionality and data structure. The **DataStructure** class can be specialized to concrete subclasses that represent different index data structures: Hashing, B-Tree, and R-Tree. Limited by our DBMS experience, we have only implemented the single dimension index at the current stage. The integration of Gist (Generalized Search Tree) index framework [HNP95] [HKP97] is in progress [NIE03].

The **Schema** class and **DatabaseFacade** class have the similar functions to their counterparts in the **LogicalLayer**. The **DatabaseFacade** class has factory methods to create the **BufferManager** and **IndexManager** objects according to the database schema. It also provides the **IQuery** interface to supports its client, the **DatabaseFacade** class in the **LogicalLayer**. The **Schema** class in the **PhysicalLayer** holds the descriptive information about the physical data storage, such as size of buffer pool, page size, index structure, etc.

The class diagram of the DBMS subsystem is shown in Figure 34. The **DBMSFacade** class provides two interfaces, **IKIAQuery**, which provides the query functionality to the external users of the DBMS; and **LAPI** to support database administration services. The **AskQuery** and **GetNext** operation in the **IKIAQuery** have the similar responsibilities to those in the **IQuery** interface, which is provided by the **DatabaseFacade** class. The **Connect_DB** and **Disconnect_DB** operation opens and closes

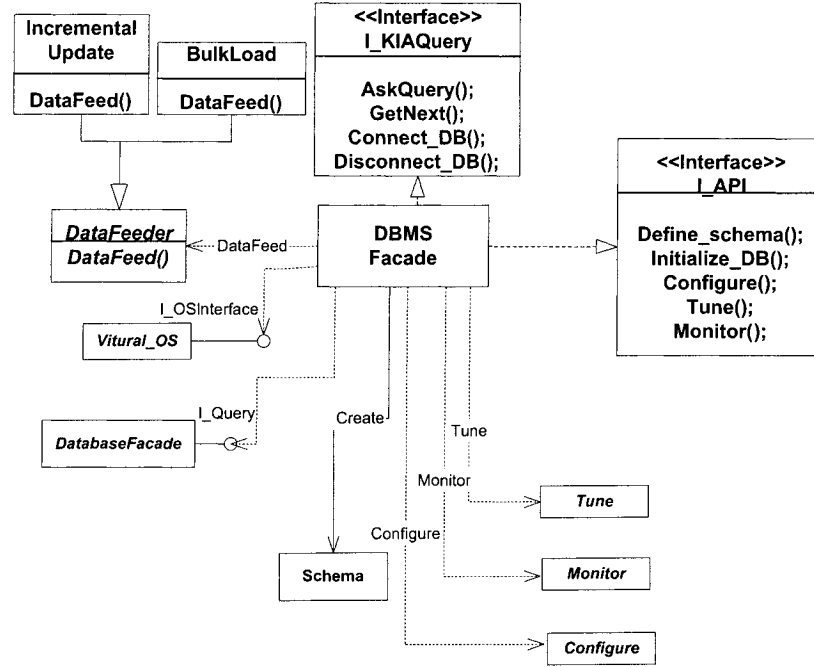


Figure 34: Know-It-All Design Model III: DBMS Subsystem

the connection to a database for query processing. The DBMSFacade class creates databases with given schemas. Query processing is accomplished by forwarding the query from the DBMSFacade class to the DatabaseFacade class in the ViewLayer. The DataFeeder abstract class has two subclasses, IncrementalUpdate and BulkLoad. The virtual function DataFeed is overridden in the two subclasses, to support different data loading mechanisms. The Virtual_OS abstract class supports the I_OSInterface to accommodate flexibility for Know-It-All to operate on various operating systems. The abstract classes Tune, Monitor, and Configure provide the Tune, Monitor, Configure services, respectively. These classes can be specialized by the framework developers to support those application-specific DBA services. By default these services are not provided by Know-It-All.

One of the most important DBMS services is to effectively access the stored data to get the information, i.e. query processing. Figure 35 shows the sequence diagram of the main activities performed when a query is issued against a database. The Database subsystem part of Figure 35 only shows the LogicalLayer. An End User uses the AskQuery operation in the DBMSFacade to pose a query. The DBMSFacade passes

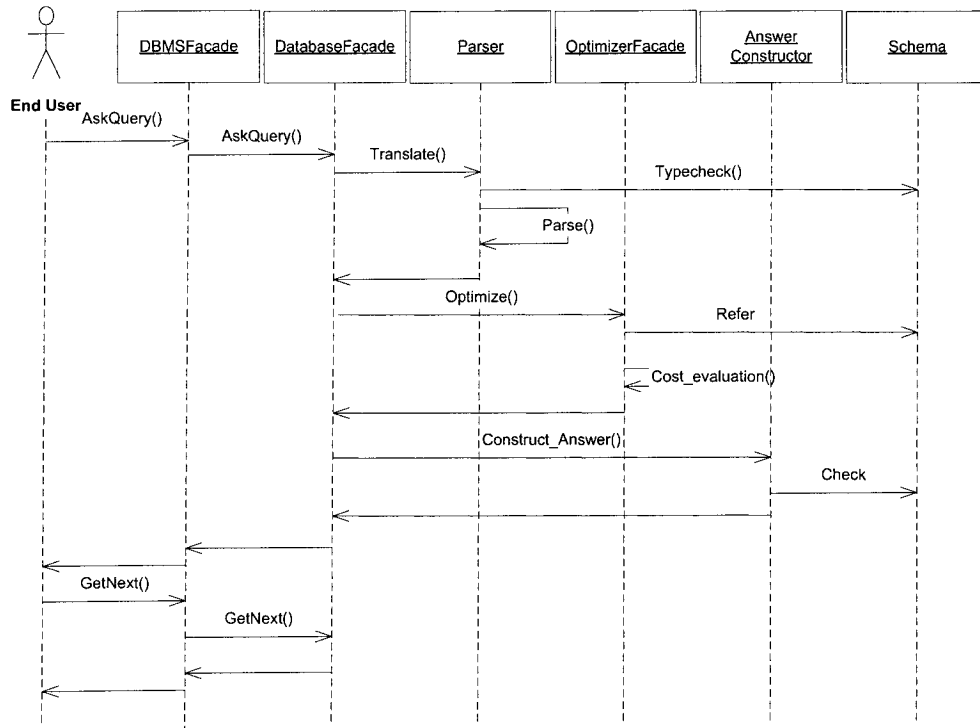


Figure 35: The Sequence Diagram of Query Processing

the query to the DatabaseFacade in the ViewLayer in the Database subsystem. Although not shown in the diagram, the query translation is performed in sequence by the ViewLayer, the ConceptualLayer, the LogicalLayer, and the PhysicalLayer, based on the four-layer architecture shown in the Know-It-All architectural model. The DatabaseFacade in the LogicalLayer translates the query by calling the Parser, then gets the optimized execution plan from the OptimizerFacade. The DatabaseFacade reconstructs the answer by calling the AnswerConstructor. The Schema assists the translation, optimization, and construction. The answer is returned in the reverse sequence of the four-layer architecture, to the DBMSFacade. The answer is represented by the reference to the root of an iterator tree. Once the answer is given to the End User, he can fetch the result by calling the GetNext operation in the DBMSFacade to return the data one by one. The DBMSFacade passes the root reference to the DatabaseFacade in the ViewLayer by calling the GetNext operation in the DatabaseFacade. The iterator in a client layer is given a reference to the corresponding iterator in its supplier layer during the AskQuery process. The GetNext operation of the DatabaseFacade in the ConceptualLayer is called by the DatabaseFacade in the ViewLayer with the reference

to the iterator (in the ConceptualLayer). The course of action is similar to that of the AskQuery operation. At the end, the Iterator (in the PhysicalLayer) points to the next DataElement in the query result is returned to the user.

4.2.5 Source Code Model

The first version prototype has been implemented with GNU C++ on the Unix system. The variability is mainly obtained by using the Standard Template Library (STL) and design pattern realization. It is not practical to put the full implementation of Know-It-All in the dissertation. Moreover, the methodology has not defined concrete map rules in source code related trace maps. Thus, source code related alignment is not discussed.

Domain		Range	
Entity	Qualifier	Entity	Qualifier
Query	Mandatory	Query	Use case
Performance	Optional	-	-
Administration	Mandatory	Administration	Base use case
DataFeed	Mandatory	DataFeed	Parent use case
Scale	Optional	-	-
Configure	Optional	Configure	Extending use case
DBConnect	Mandatory	ConnectDatabase	Included use case
DBDisconnect	Mandatory	Disconnect Database	Included use case
DBInitialize	Mandatory	Initialize Database	Included use case
DefineSchema	Mandatory	Define Schema	Included use case
Tune	Optional	Tune	Extending use case
Monitor	Optional	Monitor	Extending use case
Incremental Update	Optional	Incremental Update	Child use case
BulkLoad	Mandatory	BulkLoad	Child use case

Table 7: Entity and Qualifier Map of T_{fu}

4.3 Model Alignment Maps

The previous section describes the Know-It-All models according to the metamodels defined in chapter 3. Here, we will demonstrate how the trace maps are maintained between these models.

Domain	Range
Know-It-All : Query Composed-of relationship	-
Know-It-All : Performance Composed-of relationship	-
Know-It-All : Administration Composed-of relationship	-
Know-It-All : DataFeed Composed-of relationship	-
Know-It-All : Scale Composed-of relationship	-
Administration : Configure Composed-of relationship	Administration : Configure Extension relationship
Query : DBConnect Composed-of relationship	Query : ConnectDatabase Inclusion relationship
Query : DBDisconnect Composed-of relationship	Query : DisconnectDatabase Inclusion relationship
Administration : DBInitialize Composed-of relationship	Administration : InitializeDatabase Inclusion relationship
Administration : DefineSchema Composed-of relationship	Administration : DefineSchema Inclusion relationship
Administration : Tune Composed-of relationship	Administration : Tune Extension relationship
Administration : Monitor Composed-of relationship	Administration : Monitor Extension relationship
DataFeed : IncrementalUpdate Generalization relationship	DataFeed : IncrementalUpdate Generalization relationship
DataFeed : BulkLoad Generalization relationship	DataFeed : BulkLoad Generalization relationship

Table 8: Relationship Map of T_{fu}

4.3.1 Capability Feature Model to Use Case Model

The capability features can be divided into functional and non-functional (requirement) categories. At the current stage, the methodology does not consider the map of non-functional requirement features to the use case model. Thus, there is no corresponding entity for the **Performance** and **Scale** feature. All functional features and their relationships are mapped successfully into the use case model. The mandatory features **Query**, **Administration**, and **DataFeed** are mapped to the three use cases associated with the different actors, respectively. The mandatory sub-features of the **Query** feature are mapped to the included use case of the **Query** use case. The optional sub-features of the **Administration** feature are mapped to the extending use cases, while the mandatory sub-features are mapped to the included use cases of the **Administration** use case.

Domain		Range	
Entity	Qualifier	Entity	Qualifier
Computer	Mandatory	-	-
Network	Optional	-	-
PC	Alternative	-	-
WorkStation	Alternative	-	-
Server	Alternative	-	-
LAN	Alternative	-	-
WAN	Alternative	-	-
OSInterface	Mandatory	I_OSInterface	Interface
API	Mandatory	I_API	Interface
Windows	Alternative	-	-
OS2	Alternative	-	-
Unix	Alternative	-	-
Linux	Alternative	-	-

Table 9: Entity and Qualifier Map of T_{fa}

The Generalization relationship between the **DataFeed** feature and its child features is mapped to the Generalization relationship between the **DataFeed** use case and its child use cases. The Composed-of relationship between the **Administration** feature and its optional and mandatory sub-features is mapped to the Extension and Inclusion relationship in the use case model, respectively. The Composed-of relationship

between the Query use case and its mandatory sub-features is mapped to the Inclusion relationship. The maps conform to the map rules and satisfy the global constraints. Please refer to Figure 28, Figure 29, Table 7, and Table 8 for the details.

4.3.2 Operating Environment Feature Model to Architectural Model

The operating environment features fall into two categories: hardware interface and software interface. As stated in Section 3.2.2.3, there is no map for the hardware interface features. The software interface features and their relationships are mapped into the architectural model.

Domain	Range
HardwarePlatform: Computer Composed-of relationship	-
HardwarePlatform: Network Composed-of relationship	-
Computer: PC Composed-of relationship	-
Computer: WorkStation Composed-of relationship	-
Computer: Server Composed-of relationship	-
Network: LAN Composed-of relationship	-
Network: WAN Composed-of relationship	-
OSInterface: Windows Composed-of relationship	-
OSInterface: OS2 Composed-of relationship	-
OSInterface: Unix Composed-of relationship	-
OSInterface: Linux Composed-of relationship	-

Table 10: Relationship Map of T_{fa}

The API feature is mapped to the LAPI interface, which is required by the external software applications. The LAPI is supported by the DBMS subsystem. The OSInterface is mapped to the LOSInterface, which is provided by the hot spot subsystem Virtual_OS. The hot spot supports the flexibility of choosing different operating systems, in response to the alternative features of the OSInterface.

The Composed-of relationship between the OSInterface and its sub-features is not mapped into the range, because those sub-features are variable features, according to Rule 3 and Rule 4 of T_{fa} . Please refer to Figure 28, Figure 30, Table 9, and Table 10 for the details.

4.3.3 Domain Technology Feature Model to Design Model

The domain technology features of Know-It-All are a set of concepts, terminology, and domain specific methods in the DBMS domain. According to the rules of T_{fd} , all features are mapped to the classes in the design model. The optional and alternative features are mapped to subclasses of abstract classes, which provide the common interfaces that can be specialized according to different configurations. The cascaded refactoring methodology focuses on White-Box frameworks at the current stage, so the variability is mainly obtained through inheritance.

The mandatory features that have direct variable features are mapped to abstract classes, such as the DataModel feature. Its optional sub-feature Theory indicates the extensibility, which is maintained by the inheritance from the DataModel class to the Theory subclass. The rest of the mandatory features are mapped into classes, such as the Object, AnswerRepresentation, etc.

A Composed-of relationship is directly mapped to a class Composition relationship if the sub-feature is not a variable feature, such as the Composed-of relationship between the Index and the DataStructure feature. Otherwise, the relationship is mapped to an Inheritance relationship, such as the relationship between the Storage and Buffer feature. There are many Implemented-by relationships associated with the domain technology features. They are mapped into the Dependency relationships in the design model. For instance, the Parser feature is implemented by the Schema feature, the relationship is accordingly mapped to the Dependency relationship between the Parser and Schema class. Please refer to Figure 28, Figure 32, Table 11, and Table 12 for the details.

Domain		Range	
Entity	Qualifier	Entity	Qualifier
DataModel	Mandatory	DataModel	Abstract Class
Type	Mandatory	DataModelType	Abstract Class
Language	Mandatory	Language	Class
Theory	Optional	Theory	Class
Relational	Alternative	Relational DataModel	Class
Object	Alternative	Object DataModel	Class
Answer Representation	Mandatory	Answer Rpresentation	Class
DDL	Mandatory	DDL	Abstract Class
DML	Mandatory	DML	Abstract Class
OQL	Alternative	OQL	Class
SQL	Alternative	SQL	Class
XML	Alternative	XML	Class
Relational Operator	Mandatory	Relational Operator	Class
Schema	Mandatory	Schema	Class
Parser	Mandatory	Parser	Class
Optimizer	Mandatory	OptimizerFacade	Class
Storage	Mandatory	Storage	Abstract Class
Buffer	Optional	Buffer	Class
Index	Optional	Index	Class
Medium	Mandatory	Medium	Abstract Class
Data Structure	Mandatory	Data Structure	Abstract Class
Dimensionality	Mandatory	Dimensionality	Abstract Class
Distributed System	Alternative	Distributed System	Class
Memory	Alternative	Memory	Class
File	Alternative	File	Class
Hashing	Alternative	Hashing	Class
B-Tree	Alternative	B-Tree	Class
R-Tree	Alternative	R-Tree	Class
Single	Alternative	Single_D	Class
Multiple	Alternative	Multiple_D	Class

Table 11: Entity and Qualifier Map of T_{fd}

Domain	Range
DataModel: Type Composed-of relationship	DataModel: DataModelType Composition relationship
DataModel: Language Composed-of relationship	DataModel: Language Composition relationship
DataModel: Theory Composed-of relationship	DataModel: Theory Inheritance relationship
Language: AnswerRepresentation Composed-of relationship	Language: AnswerRepresentation Composition relationship
Language: DDL Composed-of relationship	Language: DDL Composition relationship
Language: DML Composed-of relationship	Language: DML Composition relationship
Relational: RelationalOperator Composed-of relationship	RelationalDataModel: Relational- Operator Composition relationship
Storage: Buffer Composed-of relationship	Storage: Buffer Inheritance relationship
Storage: Index Composed-of relationship	Storage: Index Inheritance relationship
Storage: Medium Composed-of relationship	Storage: Medium Composition relationship
Index: DataStructure Composed-of relationship	Index: DataStructure Composition relationship
Index: Dimensionality Composed-of relationship	Index: Dimensionality Composition relationship
Parser: Schema Implemented-by relationship	Typecheck Dependency relationship
Optimizer: Schema Implemented-by relationship	Refer Dependency relationship
Optimizer: RelationalOperator Implemented-by relationship	Use Dependency relationship

Table 12: Relationship Map of T_{fd}

4.3.4 Implementation Technique Feature Model to Source Code

For simplicity, only two features are listed in the implementation technique category, the `Iterator` and `STL`. The `STL` represents the C++ Standard Template Library (STL), which has played a very active role in `Know-It-All` implementation. Since the `Iterator` mechanism is heavily used in various places in the design, it is selected as an individual feature. For example, the `Iterator` feature can be mapped to the following code segment.

```
class Iterator{
public:
    ~Iterator();
    virtual DataElement operator*() const =0;
    virtual Iterator& operator++()=0;
    ...
};
```

So far we have verified the trace maps between the feature model and other models. Our experiences during the case study development have demonstrated that domain analysis with feature modeling is a good starting point to develop other framework models.

4.3.5 Use Case Model to Architectural Model

The use case model of `Know-It-All` classifies DBMS users into three categories: `DBA`, `Advanced User`, and `End User`. The `DBA` associates with the `Administration` use case, which describes the database management activities. The `Advanced User` is responsible for loading data into databases, and the `End User` is mainly concerned with data queries.

The `Administration` use case is mapped into the architectural model as the `LAPI` interface, which is provided by the `DBMS` subsystem. Its included use cases are mapped to the operations within the `LAPI` interface: `DefineSchema` to `Define_schema`, and `InitializeDatabase` to `Initialize_DB`. Each of those operations can be viewed as a single operation interface in order to satisfy the Rule 1 of T_{ua} , i.e. a use case is mapped to an interface.

Domain		Range	
Entity	Qualifier	Entity	Qualifier
Query	Use case	IKIAQuery	Interface
Administration	Use case	I_API	Interface
DataFeed	Use case	I_DataFeed	Interface (Operation)
Configure	Extending Use case	Configure	Hotspot (Virtual Operation)
Tune	Extending Use case	Tune	Hotspot (Virtual Operation)
Monitor	Extending Use case	Monitor	Hotspot (Virtual Operation)
Connect Database	Included Use case	Connect_DB	Interface (Operation)
Disconnect Database	Included Use case	Disconnect_DB	Interface (Operation)
Define Schema	Included Use case	Define_schema	Interface (Operation)
Initialize Database	Included Use case	Initialize_DB	Interface (Operation)
Incremental Update	Child Use case	-	(Overriden Operation)
BulkLoad	Child Use case	-	(Overriden Operation)
QE1	Episode	AskQuery	Interface (Operation)
QE2	Episode	AskQuery	Interface (Operation)
QE3	Episode	GetNext	Interface (Operation)

Table 13: Entity and Qualifier Map of T_{ua}

The extending use cases of the **Administration** are mapped to operations within the **L_API** interface. The extensibility is provided by the hot spots associated with the corresponding classes that realize those operations. For example, the **Configure** service is supported by the **Configure** class, which can be specified by its subclasses with the overridden **Configures** operation. The **DBMSFacade** class forwards the **Configure** requests to the **Configure** class. The **DataFeed** use case is mapped to the **LDatafeed** interface, which is realized by the **DataFeeder** class. It is specialized by two subclasses, the **IncrementalUpdate** and **BulkLoad**. They can override the corresponding operation to support the necessary flexibility. The overridden operation can be viewed as a single operation interface, to satisfy Rule 1, and Rule 4 of T_{ua} , i.e. the child use case must be mapped to the interface of the subclass.

Domain	Range
Configure: Administration Extension relationship	-
Tune: Administration Extension relationship	-
Monitor: Administration Extension relationship	-
ConnectDatabase: Query Inclusion relationship	Connect_DB: L_KIAQuery Composition relationship
DisconnectDatabase: Query Inclusion relationship	Disconnect_DB: L_KIAQuery Composition relationship
DefineSchema: Administration Inclusion relationship	Define_schema: L_API Composition relationship
InitializeDatabase: Administration Inclusion relationship	Initialize_DB: L_API Composition relationship
IncrementalUpdate: DataFeed Generalization relationship	IncrementalUpdate: DataFeeder Generalization relationship
BulkLoad: DataFeed Generalization relationship	BulkLoad: DataFeeder Generalization relationship

Table 14: Relationship Map of T_{ua}

The **Query** use case is mapped to the **L_KIAQuery**, which is provided by the **DBMS** subsystem. Its included use cases are mapped to the operations in the **L_KIAQuery**

interface: `ConnectDatabase` to `Connect_DB`, and `DisconnectDatabase` to `Disconnect_DB`. The episodes in the `Query` use case are mapped to the operations in the architectural model:

QE1: the `AskQuery` operation in the `LKIAQuery` supported by the DBMS subsystem

QE2: the `AskQuery` operation in the `LQuery` supported by the `ViewLayer` in the Database subsystem

QE3: the `GetNext` operation in the `LKIAQuery` supported by the DBMS subsystem

The relationships between those use cases are mapped into the architectural model according to the rules of T_{ua} . For instance, the Generalization relationship between the `DataFeed` and its child use cases is mapped to the `DataFeeder` class inheritance hierarchy. The Inclusion relationship between the `Administration` use case and its included use cases are mapped to the Composition relationship between the `LAPI` and those corresponding operations. Please refer to Figure 29, Figure 30, Table 13, and Table 14 for the details.

4.3.6 Use Case Model to Design Model

Similar to the maps between the use case model and the architectural model, the main concept of T_{ud} is to reflect the services specified by the use case model in the design decisions encapsulated by the classes and their collaborations in the design model.

It is worth noting that the `DataFeed` use case is mapped to the `DataFeed` operation, which is overridden by the subclasses of the `DataFeeder`. The `DataFeed` operation can be viewed as a single operation interface, to satisfy Rule 1 of T_{ud} . The extending use cases of the `Administration` use case are mapped to those abstract class hierarchy that supports the extensibility, by providing the virtual operations, such as the `Tune` operation in the `Tune` abstract class. The client of the `Tune` operation in the `LAPI` delegates any request to the `Tune` operation in the abstract class. The episodes in the `Query` use case are mapped to operations: QE1 is mapped to the `AskQuery` operation in the `LKIAQuery` of the DBMS subsystem; QE2 is mapped to the `AskQuery` operation in the `LQuery` of the `ViewLayer` in the Database subsystem, and QE3 is mapped to the `GetNext` operation in the `LKIAQuery`. The query processing inside the

Domain		Range	
Entity	Qualifier	Entity	Qualifier
Query	Use case	IKIAQuery	Interface
Administration	Use case	I_API	Interface
DataFeed	Use case	DataFeed	Interface (DataFeeder)
Incremental Update	Child Use case	DataFeed	Operation (IncrementalUpdate)
BulkLoad	Child Use case	DataFeed	Operation (BulkLoad)
Configure	Extending Use case	Configure	Abstract Class (Hotspot)
Tune	Extending Use case	Tune	Abstract Class (Hotspot)
Monitor	Extending Use case	Monitor	Abstract Class (Hotspot)
Connect Database	Included Use case	Connect_DB	Interface (Operation)
Disconnect Database	Included Use case	Disconnect_DB	Interface (Operation)
Define Schema	Included Use case	Define_schema	Interface (Operation)
Initialize Database	Included Use case	Initialize_DB	Interface (Operation)
QE1	Episode	AskQuery	Operation
QE2	Episode	AskQuery	Operation
QE3	Episode	GetNext	Operation

Table 15: Entity and Qualifier Map of T_{ud}

Database subsystem is mainly carried out by the interactions of those DatabaseFacade classes in the four layers.

The relationship mapping is similar to that of T_{ua} , which is explained in Section 4.3.5. For example, the Inclusion relationship between the DefineSchema and Administration, is mapped to the Composition relationship between the Define_schema operation and L_API interface. Figure 29 and Figure 32 show that the services described in the use case model are successfully mapped into the design model. Please refer to Table 15, and Table 16 for the details.

Domain	Range
Configure: Administration Extension relationship	Configure Abstract Class
Tune: Administration Extension relationship	Tune Abstract Class
Monitor: Administration Extension relationship	Monitor Abstract Class
DataFeed: IncrementalUpdate Generalization relationship	DataFeeder (DataFeed): IncrementalUpdate (DataFeed) Generalization relationship
DataFeed: BulkLoad Generalization relationship	DataFeeder (DataFeed): BulkLoad (DataFeed) Generalization relationship
Connect Database: Query Inclusion relationship	Connect_DB: I_KIAQuery Composition relationship
Disconnect Database: Query Inclusion relationship	Disconnect_DB: I_KIAQuery Composition relationship
Define Schema: Administration Inclusion relationship	Define_schema: L_API Composition relationship
Initialize Database: Administra- tion Inclusion relationship	Initialize_DB: L_API Composition relationship

Table 16: Relationship Map of T_{ud}

4.3.7 Architectural Model to Design Model

The trace map T_{ad} tallies the architectural model and design model. Both models focus on the solution of the framework, which is the most important aspect of framework development [FSJ99]. In the cascaded refactoring methodology, we emphasize maps from the layers, subsystems, and interfaces to classes and their relationships, which bridge the high level architectural design, to the low level implementation with object-oriented programming languages.

The map has to consider by far the most entities and relationships. Due to the page length limitation, maps of the entities and qualifiers are divided into two tables. Please refer to Table 17 and Table 18 for the details.

The `I>Datafeed` interface is mapped to the virtual operation `Datafeed` in the `DataFeeder` class hierarchy. The `Virtual_OS` subsystem is mapped to the `Virtual_OS` abstract class, which can be specialized to support the expected flexibility of different operating system portability. The `Configure` operation in the `LAPI` interface delegates requests to the `Configure` operation in the `Configure` abstract class, which supports the variability by inheritance and overridden functions. The `DBMS` subsystem itself is mapped to the `DBMSFacade` class, which defines the high level interfaces to provide the services of the subsystem.

The `I.Query` interface in the architectural model has multiple corresponding entities in the design model. However, all entities have the same operations (interface), which are provided by a set of `DatabaseFacade` classes from four different layers in the `Database` subsystem, respectively. Although the `KIA.Error` is only presented in the `LogicalLayer`, it exists in all other layers of the `Database` subsystem and also in the `DBMS` subsystem. For most classes in the domain, they are mapped to the corresponding abstract classes to provide the necessary flexibility. For example, the `Iterator` class is mapped to the abstract class `Iterator` in the design model. The four layers in the `Database` subsystem are mapped to the corresponding facade classes. Each of them supports the same interface and acts as the mediator within its own layer. All subsystems are mapped to the corresponding class hierarchies, which collaborate together to provide the subsystem services.

A class Composition relationship in the architectural model is mapped to the

Domain		Range	
Entity	Qualifier	Entity	Qualifier
I.KIAQuery	Interface	I.KIAQuery	Interface
I.Datafeed	Interface	DataFeed	Interface (Operation)
I.API	Interface	I.API	Interface
I.OSInterface	Interface	I.OSInterface	Interface
DataFeeder	Class	DataFeeder	Class
Incremental Update	Class	Incremental Update	Class
BulkLoad	Class	BulkLoad	Class
Configure	Class	Configure	(Abstract) Class
Tune	Class	Tune	(Abstract) Class
Monitor	Class	Monitor	(Abstract) Class
Virtual_OS	Subsystem	Virtual_OS	Class
DBMS	Subsystem	DBMSFacade	Class
AskQuery	Operation	AskQuery	Operation
GetNext	Operation	GetNext	Operation
Connect_DB	Operation	Connect_DB	Operation
Disconnect_DB	Operation	Disconnect_DB	Operation
Define_schema	Operation	Define_schema	Operation
Initialize_DB	Operation	Initialize_DB	Operation
Configure	Operation	Configure	Operation
Tune	Operation	Tune	Operation
Monitor	Operation	Monitor	Operation
I.Query	Interface	I.Query	Interface (DatabaseFacade)
Translate	Interface	Translate	Interface (Operation)
Optimize	Interface	Optimize	Interface (Operation)
Construct_Answer	Interface	Construct_ Answer	Interface (Operation)

Table 17: Entity and Qualifier Map of T_{ad} : Part I

Domain		Range	
Entity	Qualifier	Entity	Qualifier
Parser	Class	Parser	Class
Visitor	Class	Visitor	Abstract Class
Parsetree	Class	Parsetree	Class
Scanner	Class	Scanner	Class
Builder	Class	Builder	Class
SearchStrategy	Class	SearchStrategy	Abstract Class
SearchTree	Class	SearchTree	Abstract Class
DBAlgorithm	Class	DBAlgorithm	Abstract Class
Iterator	Class	Iterator	Abstract Class
KIA_Error	Class	KIA_Error	Abstract Class
Schema	Class	Schema	Class
Relation	Class	Relation	Abstract Class
Attribute	Class	Attribute	Abstract Class
Relational Operator	Class	Relational Operator	Class
Answer Constructor	Class	Answer Constructor	Class
Index	Class	Index	Class
Buffer	Class	Buffer	Class
DataContainer	Class	DataContainer	Abstract Class
DataElement	Class	DataElement	Abstract Class
Page	Class	Page	Abstract Class
ViewLayer	Layer	DatabaseFacade (from ViewLayer)	Class
ConceptualLayer	Layer	DatabaseFacade (from ConceptualLayer)	Class
LogicalLayer	Layer	DatabaseFacade (from LogicalLayer)	Class
PhysicalLayer	Layer	DatabaseFacade (from PhysicalLayer)	Class
Parser	Subsystem	Parser	Class (hierarchy)
Optimizer	Subsystem	OptimizerFacade	Class (hierarchy)
IndexManager	Subsystem	IndexManager	Class (hierarchy)
BufferManager	Subsystem	BufferManager	Class (hierarchy)
FileManager	Subsystem	DataContainer	Class (hierarchy)
AskQuery	Operation	AskQuery	Operation
GetNext	Operation	GetNext	Operation

Table 18: Entity and Qualifier Map of T_{ad} : Part II

Domain	Range
Schema: Relation Composition relationship	Schema: Relation Composition relationship
Schema: Attribute Composition relationship	Schema: Attribute Composition relationship
DataContainer: DataElement Composition relationship	DataContainer: DataElement Composition relationship
DBMS: Database Dependency relationship	DBMSFacade: DatabaseFacade Dependency relationship
ViewLayer: ConceptualLayer Dependency relationship	DatabaseFacade: DatabaseFa- cade Dependency relationship
ConceptualLayer: LogicalLayer Dependency relationship	DatabaseFacade: DatabaseFa- cade Dependency relationship
LogicalLayer: PhysicalLayer Dependency relationship	DatabaseFacade: DatabaseFa- cade Dependency relationship
DBMS: I_API Realization relationship	DBMSFacade: I_API Realization relationship
DBMS: I_KIAQuery Realization relationship	DBMSFacade: I_KIAQuery Realization relationship
Virtual_OS: I_OSInterface Realization relationship	Virtual_OS: I_OSInterface Realization relationship
Database: I_Query Realization relationship	DatabaseFacade: I_Query Realization relationship
Parser: Translate Realization relationship	Parser: Translate Realization relationship
Optimizer: Optimize Realization relationship	OptimizerFacade: Optimize Realization relationship
AnswerConstructor: Con- struct_Answer Realization relationship	AnswerConstructor: Con- struct_Answer Realization relationship
DataFeeder: IncrementalUp- date Generalization relationship	DataFeeder: IncrementalUp- date Generalization relationship
DataFeeder: BulkLoad Generalization relationship	DataFeeder: BulkLoad Generalization relationship

Table 19: Relationship Map of T_{ad}

same relationship in the design model. A Dependency relationship between the subsystems or layers is mapped to a Dependency relationship between classes. For example, the Dependency between the DBMS subsystem and the Database subsystem is mapped to the Dependency between the DBMSFacade and the DatabaseFacade from the ViewLayer, because the ViewLayer is the direct supplier to the DBMS subsystem. The rationale is also valid to the map of the Realization relationships, which are mapped into the range as the Realization relationships between interfaces and classes. Please refer to Table 19 for the details.

4.4 Model Refactorings

The cascaded refactoring methodology views framework evolution as framework refactoring followed by framework extension. The methodology focuses on framework refactoring at the current stage. It addresses the issue of identification and preservation of consistency through the refactorings, i.e. how the impact of changes on the design and implementation due to the alteration of requirements is illustrated and managed with the trace maps. The solution proposed by the methodology is demonstrated in this section with two example refactorings on the **Know-It-All** models. The first example changes the **Configure** service to be a common property of **Know-It-All**. The second example alters the database creation service from a frozen spot to a hot spot.

4.4.1 Example 1

The **Configure** service for DBAs is responsible for database configuration. It is a variable feature in the original **Know-It-All** feature model. It is possible to change it to be a mandatory property, i.e. common to all **Know-It-All** applications. Furthermore, the change should be propagated to all other models. The cascade of refactorings is summarized in Figure 36. The following subsections will treat each model in turn.

Problem: Change the **Configure** feature from an optional feature to a mandatory feature of the **Know-It-All** feature model, and cascade the refactoring to other models.

Solution: The refactoring starts at the capability feature category. So the following refactoring path should be followed (see Section 3.3.1):

Capability Feature refactoring path:

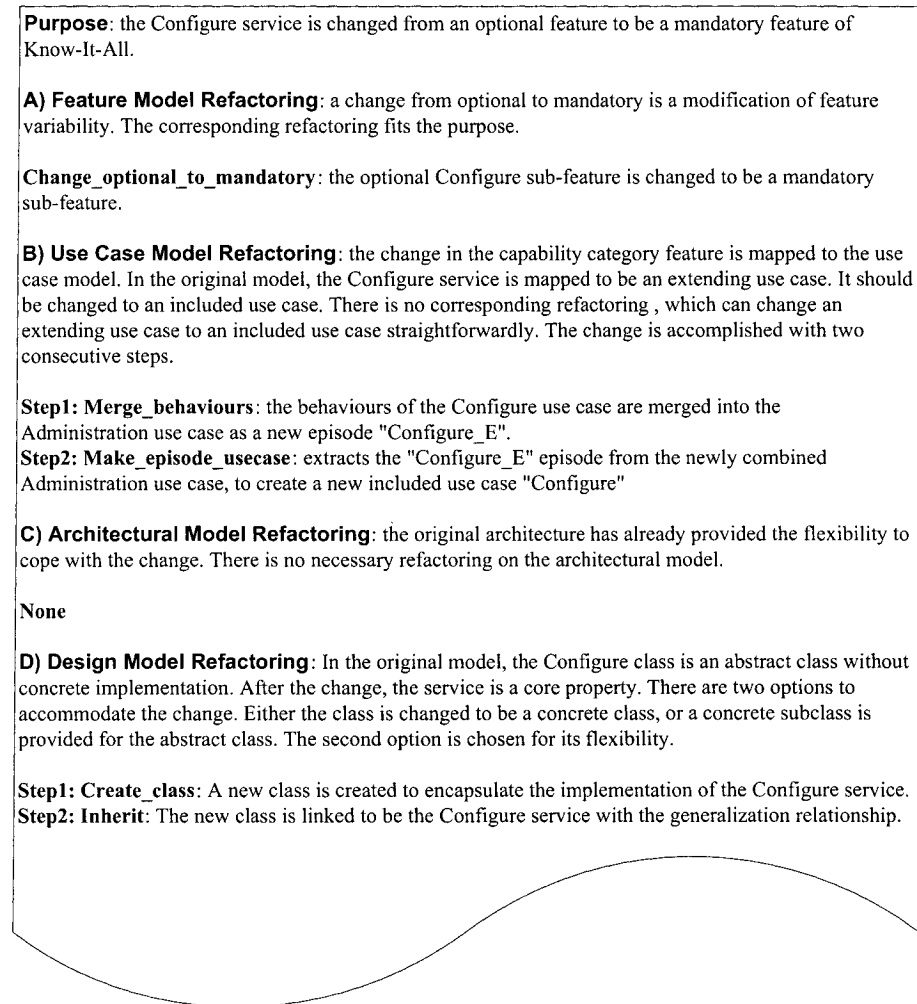


Figure 36: Roadmap of Refactorings in Example One

1 $M_f \Rightarrow M_u: T_{fu}$

2 $M_u \Rightarrow M_a: T_{ua}$

3 $M_a \Rightarrow M_d: T_{ad}$

4 $M_d \Rightarrow M_i: T_{di}$

Roadmap: See Figure 36.

4.4.1.1 Feature Model Refactoring

A change from optional to mandatory is a modification of feature variability. It is sensible to choose the **Change_optional_to_mandatory**. Next, we need to prove that the refactoring preserves the “behaviour”.

According to the cascaded refactoring methodology, the preserved “behaviour” of feature model refactorings is the collection of valid feature sets, each of which specifies an existing application that has been created from the product line. Prior to the refactoring, the **Configure** feature is an optional feature, which may or may not be present in a feature set of the **Know-It-All** feature model. However, the **Configure** feature must belong to all feature sets of the existing applications in order to satisfy the precondition of the refactoring. After the change, the feature is mandatory in the post-refactoring feature model, that is, all feature sets of the model should have the **Configure** feature. Thus, all legacy code of the existing applications before the refactoring still conforms to the feature model. Since the **Configure** feature is the only argument taken by the refactoring, and its change does not invalidate any feature set of existing applications, the “behaviour” is preserved. The decision record is shown in Figure 37.

Decision Record 1:
Intent: The Configure service is changed to be a mandatory feature.
Choice: Change_optional_to_mandatory
Arguments: Configure feature
Validation: The Configure feature becomes a mandatory feature. To satisfy the pre-condition of the refactoring, any feature set that specifies an existing application built from the Know-It-All framework must have the Configure feature before the refactoring. Thus, the change does not invalidate any existing feature set prior to the refactoring. The behaviour is preserved.

Figure 37: Feature Model Refactoring: Decision Record 1

The new feature model is shown in Figure 38. The **Configure** feature is changed to be a mandatory feature.

4.4.1.2 Use Case Model Refactoring

The refactoring of the feature model has changed the **Configure** feature from an optional feature to a mandatory feature. The impact of the feature model refactorings should be translated via the trace map T_{fu} to determine the constraints on the use case model. In the original use case model, the **Configure** use case is an extending use case of the **Administration** use case, it has to be an included use case of the **Administration** after the change, according to Rule 4 of T_{fu} .

There is no use case refactoring that can directly change an extending use case to an included use case. Thus, the task has to be accomplished by multiple refactorings. We view the change as “elimination of the extending use case” followed by “creation of the included use case”. Two refactorings: **Merge_behaviours** and **Make_episode_usecase** are chosen based upon this idea.

The methodology defines the episode set as the preserved “behaviour” of the use case model refactoring. The post-refactoring use case model should preserve the set of episodes of the pre-refactoring use case model. An Extension relationship in a use case model indicates that the behaviours defined in the base use case can be extended by the behaviour defined in the extending use case, once the extension condition is satisfied. Since the **Configure** feature has become a mandatory sub-feature of the **Administration** feature in the feature model, the **Configure** service should be a persistent part of the **Administration** service. Also, the set of episodes contained in the **Configure** and **Administration** use cases must be able to be combined without violating their coherence, in order to satisfy the preconditions of **Merge_behaviours**. Thus, we conclude that the extension condition is always satisfied in the pre-refactoring use case model. Since the only difference between the pre- and post-refactoring models is the relationship between the **Configure** and **Administration** use case, the “behaviour” is preserved during the refactorings. The decision records are given in Figure 39 and Figure 40.

The first refactoring merged the behaviours in the **Configure** use case as a new episode **Configure_E** into the (new) **Administration** use case, and the **Configure** use case disappeared. Since the (new) **Administration** use case was created by the combination

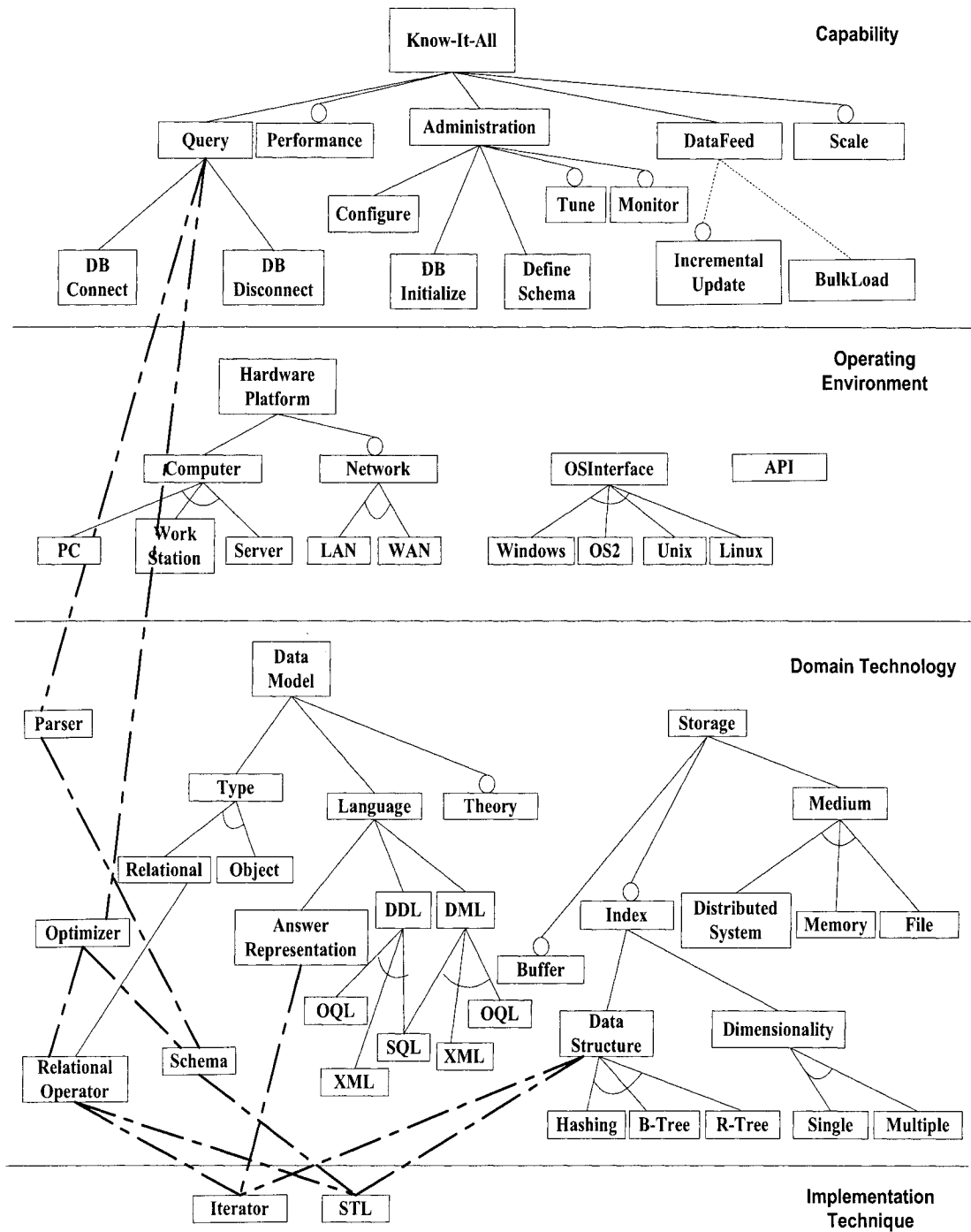


Figure 38: Modified Know-It-All Feature Model

Decision Record 1:

Intent: Merge the behaviours of the Configure use case and Administration use case.
Choice: Merge_behaviours
Arguments: Configure use case, Administration use case, Administration_1 use case
Validation: The behaviours of the two use cases are merged into a new use case Administration_1. The behaviours of the Configure use case are described by the Configure_E episode. The Configure and Administration use case disappear. To simplify the situation, the Administration_1 use case is renamed to Administration after the merge. This does not cause any name clash.

Figure 39: Use Case Model Refactoring: Decision Record 1

Decision Record 2:

Intent: Make the Configure_E episode to be an included use case, called Configure.
Choice: Make_episode_usecase
Arguments: Configure_E episode, Administration use case, Configure use case.
Validation: A new use case, called Configure, is created with the behaviour specified by the Configure_E episode. An inclusion link is added between the Configure and Administration use cases. There will be no name clash because the Configure use case does not exist before the refactoring.

Figure 40: Use Case Model Refactoring: Decision Record 2

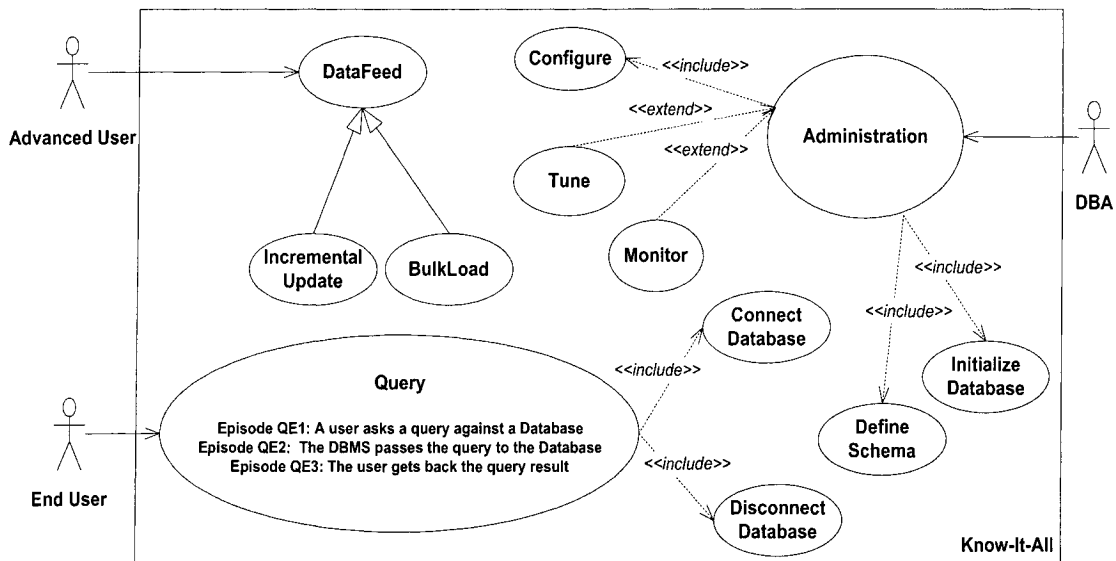


Figure 41: Modified Know-It-All Use Case Model

of the episodes from the (old) **Administration** and (old) **Configure** use case. The removal of the **Configure_E** should not cause any problem because other episodes in the (new) **Administration** use case are independent to **Configure_E**. Furthermore, the action does not make any name clash because the **Configure** use case was deleted by the first refactoring. Thus, the precondition of the second refactoring is satisfied and the **Make_episode_usecase** can be performed. The **Configure_E** episode is taken out from the **Administration** use case and a new use case with the name of “Configure” is created. The only thing affected by the second refactoring is the **Administration** and **Configure** use case. The set of episodes of these two use cases are preserved. Figure 41 shows the modified use case model. The **Configure** use case is changed to be an included use case of the **Administration** use case.

4.4.1.3 Architectural Model Refactoring

The refactoring on the use case model has changed the **Configure** use case from an extending use case to an included use case of the **Administration** use case, i.e. the variability of the **Configure** use case is changed from the Extension to the Inclusion relationship. The task on the architectural model refactoring is to change the original model to satisfy the constraints that is translated via T_{ua} .

In the pre-refactoring model, there is a hot spot associated with the **Configure** operation. The hot spot is achieved through the **Configure** class (hierarchy). Applications customized from **Know-It-All** can decide whether to have the **Configure** service by altering the **Configure** class specialization, i.e. whether to supply the actual **Configure** service. Inspection on the architecture model shown in Figure 30 has found that the **Configure** operation is included in the **LAPI** interface. It conforms to the Inclusion relationship map rule of T_{ua} . Therefore, the pre-refactoring architectural model has already provided the flexibility to accommodate the change. It is not necessary to change the architectural model.

4.4.1.4 Design Model Refactoring

There is no explicit constraint on the design model refactoring since no change has been made on the architectural model. However, the **Configure** class is an abstract class without concrete implementation in the original design model. The service becomes to a common property according to the requirement. Two options are available

to deal with the change: make the `Configure` abstract class to be a concrete class, or provide a concrete subclass for the `Configure` abstract class. The second option is chosen for its flexibility.

<p>Decision Record 1:</p> <p>Intent: Create a new class.</p> <p>Choice: <code>Create_class</code></p> <p>Arguments: <code>ConfigureImp</code> class</p> <p>Validation: A <code>ConfigureImp</code> class is created. It has no relationship with any other class, and has no operation either. A new name is picked up so there is no name clash. The precondition of the refactoring is satisfied.</p>
--

Figure 42: Design Model Refactoring: Decision Record 1

The task cannot be accomplished by a single design model refactoring since such a refactoring is not available. However, we can view the task as “create a new class” followed by “make the new class a subclass of the `Configure` class”. Thus, two refactorings are chosen: `Create_class` and `Inherit`. The decision records are given in Figure 42 and Figure 43.

<p>Decision Record 2:</p> <p>Intent: Make the <code>ConfigureImp</code> a subclass of the <code>Configure</code> abstract class.</p> <p>Choice: <code>Inherit</code></p> <p>Arguments: <code>ConfigureImp</code> class, <code>Configure</code> class</p> <p>Validation: The <code>ConfigureImp</code> class becomes a subclass of the <code>Configure</code> class. Since the <code>ConfigureImp</code> class has neither parent class nor subclass before the refactoring, as stated in Decision Record 1, the precondition of the refactoring is satisfied. The <code>Configure</code> virtual function in the <code>Configure</code> class must be overridden by the <code>ConfigureImp</code> class to provide the default <code>Configure</code> service,</p>

Figure 43: Design Model Refactoring: Decision Record 2

The first refactoring creates a new class `ConfigureImp`. The second refactoring adds a Generalization relationship between the `ConfigureImp` and `Configure` class. The variability is achieved through the class hierarchy.

Since the change is localized in the DBMS subsystem, we only show this part of the modified design model in Figure 44. The `ConfigureImp` class is created to be a subclass of the `Configure` abstract class.

In the post-refactoring design model, the `Configure` operation within the `IAPI` interface is provided by the `DBMSFacade` class, which depends on the `Configure` class hierarchy. This conforms to Rule 3 of T_{ud} , i.e. the Inclusion relationship in the

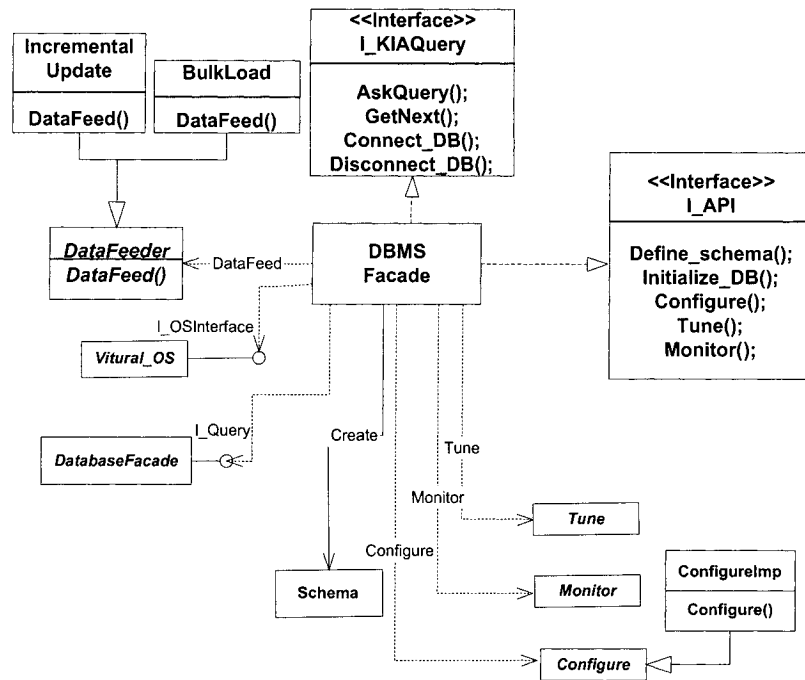


Figure 44: Modified Design Model

use case model is mapped to the Dependency relationship in the design model. The change from the requirements is successfully cascaded to the design of the framework.

4.4.2 Example 2

The **Know-It-All** applications support database creation service by the DBMS subsystem. Multiple database creation is a desirable property, i.e. **Know-It-All** leaves the decision of which database to create to be made by the application developers, instead of the framework developers. Changes due to the requirement alteration should be propagated to all affected models. The cascade of refactorings is summarized in Figure 45. The following subsections will treat each model in turn.

Problem: Modify Know-It-All to make it able to create different databases.

Solution: The refactoring starts at the capability feature category. So the following refactoring path should be followed (see Section 3.3.1):

Capability Feature refactoring path:

- 1 $M_f \Rightarrow M_u: T_{fu}$
- 2 $M_u \Rightarrow M_a: T_{ua}$

3 $M_a \Rightarrow M_d: T_{ad}$

4 $M_d \Rightarrow M_i: T_{di}$

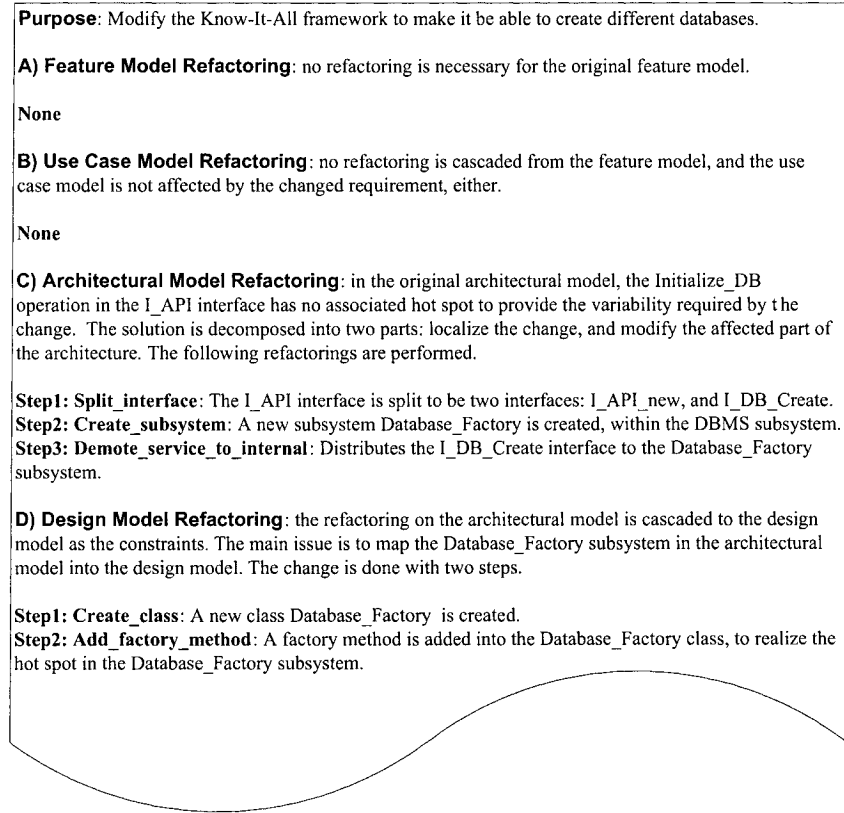


Figure 45: Roadmap of Refactorings in Example Two

Roadmap: See Figure 45.

The requirements of Know-It-All are changed, so, refactorings should begin with the capability feature model and follow its refactoring path. However, the only feature related to the database creation is the DBInitialize mandatory sub-feature. Neither the variability nor the relationships to other features of the DBInitialize feature has to be changed due to the new requirement. Therefore, no refactoring is necessary for the feature model. The database creation service is represented by the InitializeDatabase use case in the use case model. It is an included use case of the Administration use case (Figure 29). The new requirement has no impact on the use case model either. Thus, the refactorings start with the architectural model.

4.4.2.1 Architectural Model Refactoring

The main issue addressed by the refactorings is to provide the required variability of object creation. The framework provides the database creation interface, but lets the subclasses in the applications decide which database to instantiate. The cascaded refactoring methodology handles the variability in the architectural model by hot spots and design patterns (Section 3.2). As stated in the design pattern book [GOF94], the Factory Method design pattern is applicable for this context. The intuitive idea is to change the Initialize_DB operation to be a factory method. However, such an architectural refactoring is not available. Inspired by the Hot Spot Generalization Approach [SCHM97], we want to create a new subsystem to host the Factory Method design pattern and associate the subsystem to the identified hot spot. The following steps are carried out:

Decision Record 1:

Intent: Split the I_API interface to be two new interfaces: I_API_New, and I_DB_Create.

Choice: Split_interface

Arguments: I_API interface, I_DB_Create interface, I_API_New interface, DBMS subsystem

Validation: The I_API interface is split into two new interfaces: I_DB_Create and I_API_New. The operations in the I_API interface are distributed to the new interfaces. The Initialize_DB operation is assigned to the I_DB_Create, the rest of the operations are put into the I_API_New. All services provided by the I_API are supported by either the I_DB_Create, or the I_API_New interface. There is no name clash. The precondition is satisfied.

Figure 46: Architectural Model Refactoring: Decision Record 1

1. Split the I_API interface to be two interfaces: I_DB_Create and I_API_New.
2. Create a hot spot subsystem Database_Factory inside the DBMS subsystem
3. Redistribute the I_DB_Create from the DBMS to the Database_Factory subsystem.

Based upon the above idea, three refactorings are chosen: **Split_interface**, **Create_subsystem**, and **Demote_service_to_internal**. The decision records of each of them are given in Figure 46, Figure 48, and Figure 50.

After the first refactoring, the DBMS subsystem in the modified architectural model is given in Figure 47. The Initialize_DB operation is put into the I_DB_Create interface. Any possible change related to the operation is localized into the interface.

The second refactoring creates a new subsystem, called Database_Factory, which does not clash with existing names inside the DBMS subsystem. It does not associate with any interface or operation.

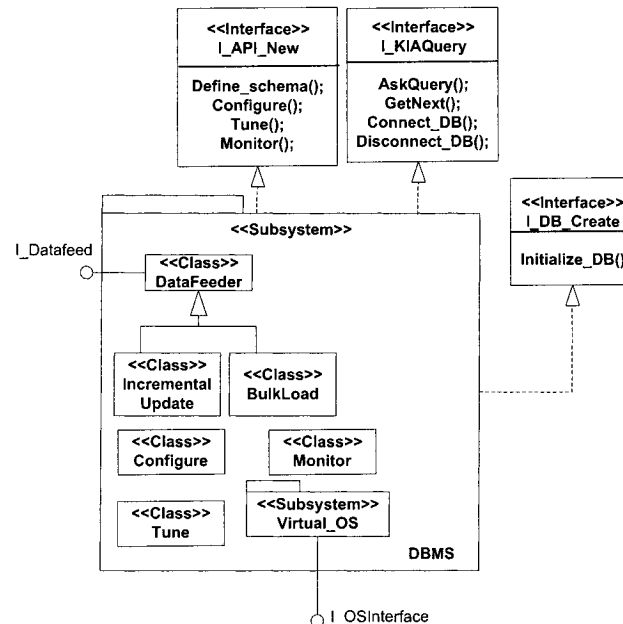


Figure 47: Architectural Refactoring I

Figure 49 presents the modified DBMS subsystem after the second refactoring. The Database_Factory subsystem is created inside the DBMS subsystem.

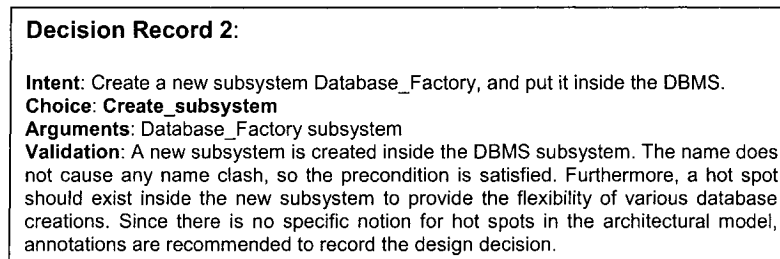


Figure 48: Architectural Model Refactoring: Decision Record 2

After the third refactoring, the flexibility required by the changed requirement is provided by the Database_Factory hot spot subsystem with the I.DB_Create interface (shown in Figure 51).

The cascaded refactoring methodology takes the set of services provided by the framework as the preserved “behaviour” of architectural model refactoring. We need

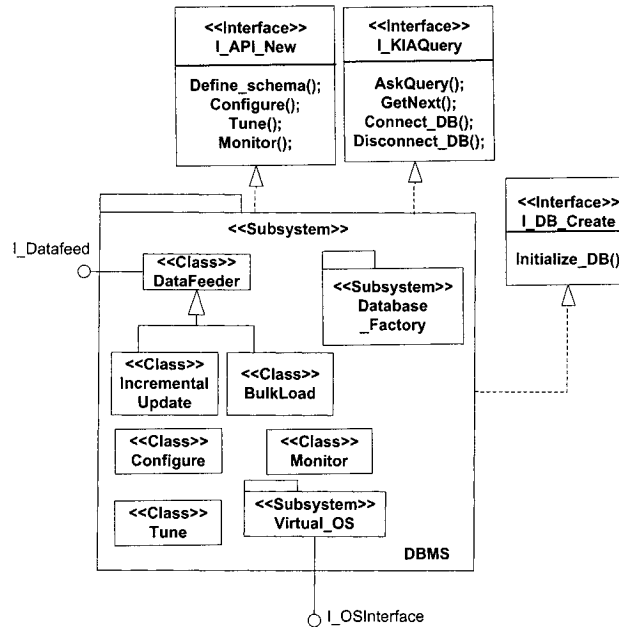


Figure 49: Architectural Refactoring II

to prove that the post-refactoring architectural model provides all the services of the pre-refactoring model.

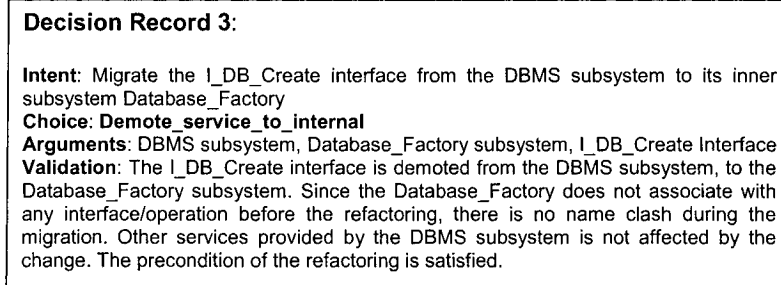


Figure 50: Architectural Model Refactoring: Decision Record 3

In this example, the only difference between the pre- and post-refactoring architectural model is the variability of the database creation service. The hot spot brought by the refactorings “augments” the original service with better flexibility. In fact, the set of services of the post-refactoring model is a superset of that of the pre-refactoring model. Any service of the pre-refactoring model is provided by the post-refactoring model. So, the “behaviour” is preserved by the refactorings.

We want to verify whether the architecture related trace maps are still maintained after the changes on the architecture. The trace map T_{fa} is concerned with the Operating Environment features and the interfaces in the architectural model. Prior to the architectural refactorings, the API feature was mapped to the L_API interface (see Table 9). Refactorings on the architectural model split the L_API interface into the L_DB_Create and L_API_New interfaces. Thus, the API feature should be mapped to the L_API_New interface. The L_DB_Create interface does not affect T_{fa} because trace maps are not symmetric (page 116).

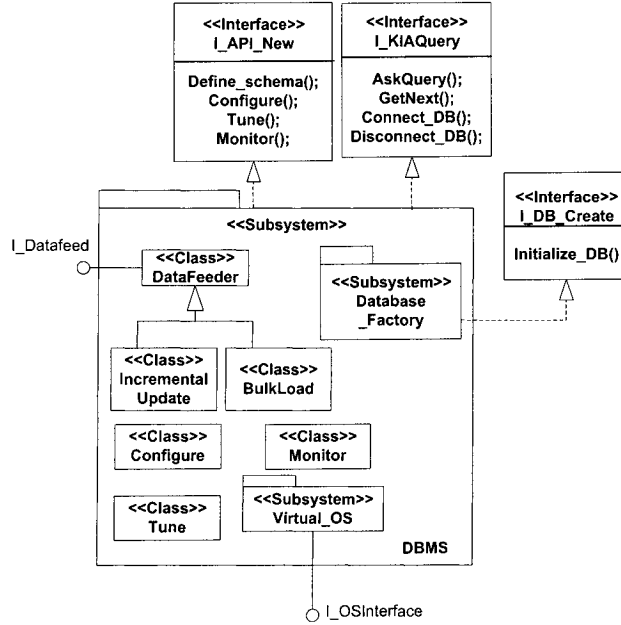


Figure 51: Architectural Refactoring III

The trace map T_{ua} is concerned with the use cases and the interfaces in the architectural model. Previously, the Administration use case was mapped to the L_API interface, its included use case InitializeDatabase was mapped to the Initialize_DB operation (see Table 13), and the Inclusion relationship was mapped to the Composition relationship between the Initialize_DB operation and L_API interface (see Table 14). To maintain the trace map after the architectural refactorings, the Administration use case is mapped to the L_API_New interface. The InitializeDatabase use case is mapped to the L_DB_Create interface. The Inclusion relationship is mapped to the Dependency relationship from the L_API_New to L_DB_Create. This does not violate either Rule 5 of T_{ua} nor the architectural metamodel, because the Database.Factory

subsystem is included in the DBMS subsystem.

Since there is no change in the use case model, the trace map T_{fu} is not affected and no issue arises in the Capability feature category.

4.4.2.2 Design Model Refactoring

According to the trace map T_{ad} , the design decisions encapsulated in the architectural model must exist in the design model. Impact of refactorings on the architectural model becomes the constraints of the refactorings for the design model.

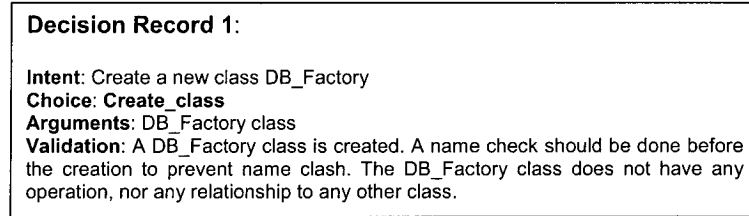


Figure 52: Design Model Refactoring: Decision Record 1

Two issues are taken into consideration during the design model refactorings: 1) the new subsystem **Database_Factory**; 2) the variability provided by the hot spot. The first issue is addressed by adding a class (hierarchy) to support the services of the **Database_Factory** subsystem. Since the hot spot is realized by the Factory Method design pattern in the architectural model, the same idea is also used in the design model. Two refactorings are chosen from Tokuda's design refactorings [TOKU99]: **Create_class** and **Add_factory_method**. The decision records are given in Figure 52 and Figure 54.

After the first refactoring, a new class, called **DB_Factory**, is created. However, it has no operation and does not provide any interface. The modified part of the design model is shown in Figure 53.

The **Add_factory_method** has four arguments [TOKU99]:

1. **Factory:** the class into which the factory method is added
2. **Product:** the class of the object which is created by the factory
3. **Ptype:** the return type of the factory method. Ptype must be either the Product or a superclass of the Product

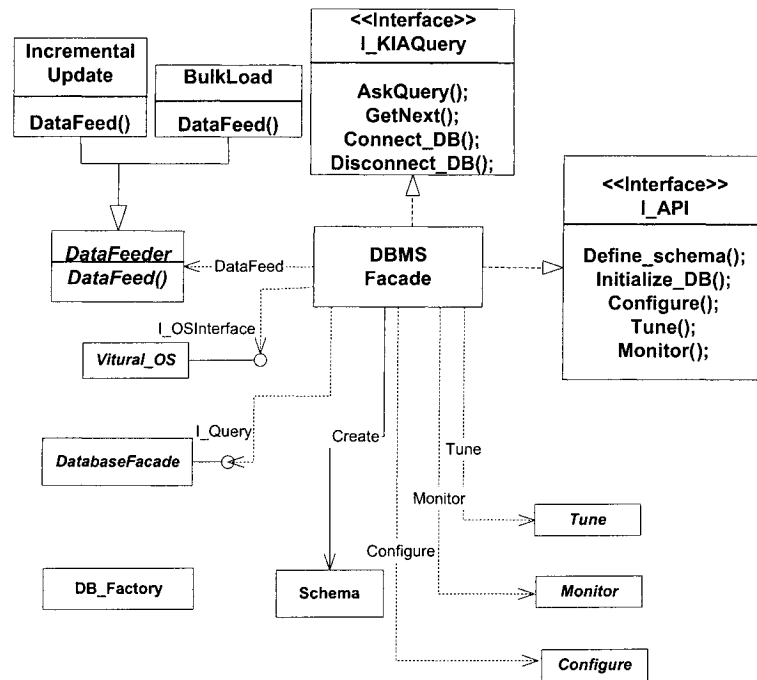


Figure 53: Design Model Refactoring I

4. method: the name of the method which is added

A method is a virtual member function [STRO97]. An operation is a service that is described as a signature with a name and parameters [OMG03]. We view a service in terms of interface and operations in the design model. An operation can be a concrete operation or a virtual operation. The difference between a method and an operation is not distinguished in the methodology.

Decision Record 2:

Intent: Create a factory method in the DB_Factory class

Choice: Add_factory_method

Arguments: DB_Factory class, Initialize_DB() method, DatabaseFacade class, DatabaseFacade* type

Validation: A factory method Initialize_DB() is created inside the DB_Factory class. A dependency relationship is added from the DB_Factory class to the DatabaseFacade class, because the Product of the factory method is a DatabaseFacade object. The return type of the factory method is DatabaseFacade*. The factory is the DB_Factory class. The variability is realized by the Factory design pattern. Since there is no operation inside the DB_Factory class before the refactoring, the Initialize_DB method should not cause any name clash. The precondition is satisfied.

Figure 54: Design Model Refactoring: Decision Record 2

The DB_Factory class provides the flexibility with the added factory method after

the second refactoring. It realizes the Database_Factory subsystem in the architectural model. Figure 55 presents the modified part of the design model after the second refactoring.

The comparison on the pre- and post-refactoring architectural model brings out another issue: the I_API interface is decomposed to be two new interface I_DB_Create, and I_API_New. The I_DB_Create interface is provided by the new subsystem Database_Factory. The elements I_DB_Create interface, I_API_New interface, and the Database_Factory subsystem should be mapped to the corresponding entities in the design model. In addition, the Inclusion relationships between the DBMS to Database_Factory subsystem has to be mapped to a Dependency relationship. To maintain the trace maps between the architectural model and the design model, more changes have to be made on the design model.

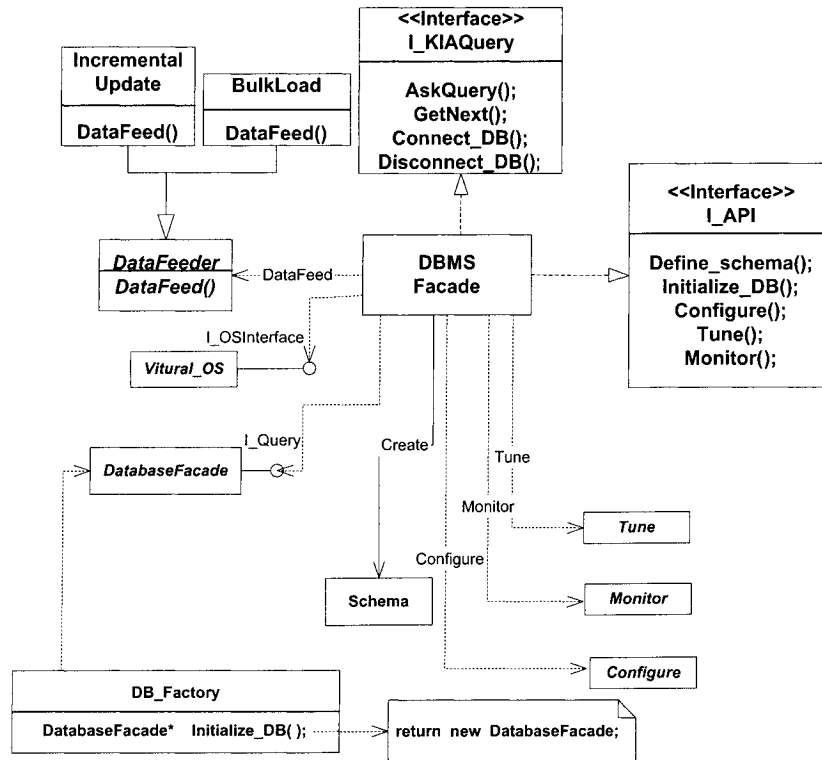


Figure 55: Design Model Refactoring II

Figure 56 shows the modified part of the final design model. The I_API_New and I_DB_Create interfaces are created and associated to the corresponding classes. The Database_Factory subsystem is mapped to the DB_Factory class. The Inclusion

relationship between the DBMS and Database_Factory subsystem is mapped to the Dependency relationship from the DBMSFacade to DB_Factory class. The Realization relationship between the Database_Factory subsystem and I_DB_Create interface is mapped to the Realization relationship between the DB_Factory class and I_DB_Create interface. However, there are no design model refactorings available for those changes, to our best knowledge.

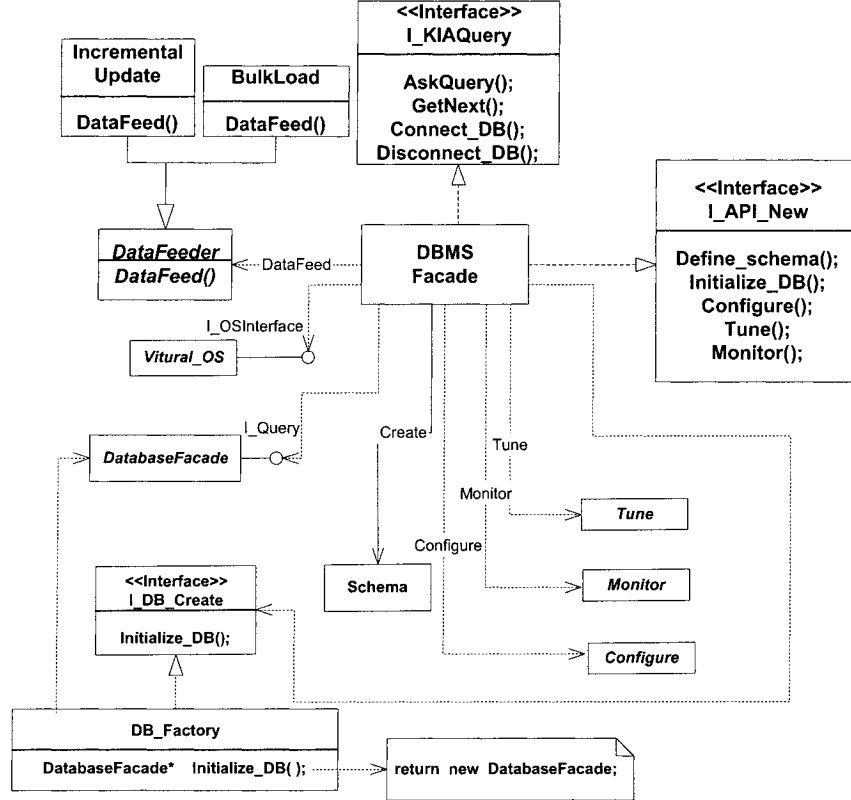


Figure 56: Design Model Refactoring III

Another solution for this issue, is to perform more refactorings on the architectural model shown in Figure 51. The rules of T_{ad} are still the constraints, though they are used in a “reverse” way. The process is illustrated as follows:

1. The **I_API_New** interface has to be renamed to the **I_API** interface, which is provided by the **DBMS** subsystem. The **DBMS** subsystem corresponds to the **DBMSFacade** class in the design model. This change can be done with the **Change_interface_name** architectural model refactoring. Since there is no

L_API interface before the refactoring, there will be no name clash. Furthermore, the L_API_New interface does not have any client. Thus, the preconditions are satisfied, the argument L_API_New interface is renamed to the L_API interface after the refactoring.

2. The DB_Factory class in the design model (Figure 55) has the Initialize_DB operation, which can be viewed as a single operation interface, to match the L_DB_Create interface provided by the Database_Factory subsystem. The subsystem corresponds to the DB_Factory class.
3. The L_API interface in the design model (Figure 55) includes the Initialize_DB operation, which is not in the L_API interface in the architectural model. However, this does not violate Rule 1 of T_{ad} , because the L_API interface in the design model provides *all* services specified by the L_API interface in the architectural model.

The rationale of the process is Rule 1 and Rule 4 of T_{ad} , i.e. interface mapping and subsystem mapping. The only refactoring performed during the whole process renamed an interface. Thus, we conclude that services of the architectural model are not changed during the refactoring. The “behaviour” is preserved.

4.5 Discussion

We present the feature model, use case model, architectural model, and design model of Know-It-All. The concrete entity and relationship mapping of each trace map is described. The example refactorings demonstrate how the refactorings are cascaded on the models via the trace maps to achieve refactoring of a framework.

Issues related to the methodology have been validated with the Know-It-All case study:

1. Whether the chosen notion of behaviour is appropriate for each kind of model.

Refactoring is a behaviour-preserving process. Design and source code refactorings take functionality as the preserved behaviour. After we extend the concept to the feature model, use case model, and architectural model, it is important to

define the notion of behaviour for those models as the invariants during refactorings. The concrete trace maps and cascaded refactoring examples demonstrate the chosen model behaviour meets our initial expectation, that is, specify the main property that must be preserved during the transformation of each of the model. For example, a feature set is composed of features that are chosen from a feature model. An application of a product line is specified by a feature set that is derived from the feature model of the product line. Thus, feature sets of all existing applications should conform to the post-refactoring feature model. It is appropriate to use the collection of feature sets to be the notion of behaviour for a feature model, as demonstrated in example 1. On the other hand, we are only concerned with “functionality” as the default invariants for the use case model and the architectural model. Other quality attributes such as performance and variability should be considered in future work.

2. Whether the chosen architectural notion is appropriate for framework specification.

Limited by our experiences on software architecture, we only focus on the logical view and define the metamodel in terms of layers, subsystems, classes, and interfaces, to follow the convention in the literature. The metamodel is closely related to the UML design models, in order to facilitate the alignment mappings. For example, a generalization relationship in an architectural model is always mapped to a generalization relationship in a design model. The entity and relationship mappings of trace map T_{ad} (see Table 17, Table 18, and Table 19) demonstrates the ease of mappings between the architectural model and design model. Furthermore, the metamodel stresses on a hierarchical view of architecture to reduce complexity and improve flexibility, as shown in the **Know-It-All** architectural model (see Figure 30). On the other hand, runtime entity and physical deployment information should be added to increase applicability of the metamodel. We also feel it beneficial to have special notion of hot spots in architectural models to specify variability, to facilitate description of trace maps, and to assist framework documentation.

3. Will the refactorings maintain the internal traceability and alignment for each kind of model.

The traceability amongst the models is described using trace maps, which are concerned with not only the horizontal traceability, but also correspondence of commonality and variability of the set of models. The methodology views refactoring of framework as two stages in terms of granularity (see Section 3.3.1). The internal traceability (vertical) of each model is specified by its metamodel, and kept by the refactorings of that individual model. For example, the set of episodes is chosen as the preserved behaviour for use case model refactorings, as described in example 1. The alignment between the models is obtained by the cascaded refactorings via trace maps. Furthermore, although the use case model and architectural model only treat functionality as the preserved behaviour, commonality and variability are stressed with the trace map rules, and considered during the refactorings. It is worth noting that name tracing is also useful as implicit traceability to assist the mappings, such as the **Administration** feature and the **Administration** use case. On the other hand, the map rules are only applicable on subsets of those models. The lists of refactorings of each model are also limited. Future work on them is expected.

4. Whether the document template is appropriate to record the refactoring for each kind of model.

Refactoring is viewed as an issue-driven activity in the methodology. The issues are represented as a collection of decisions, each of which is described by a decision record. The refactoring examples demonstrate that a set of decision records specify the road map of refactoring of a framework. The template is appropriate to record the refactoring for the models in terms of the notion of behaviour. However, it may need modification once other quality attributes are considered during refactoring of those models.

As a conclusion, the cascaded refactoring methodology is able to:

- Specify frameworks with a set of models across the analysis, design, and implementation

The requirements are captured in a feature model and a use case model, design in architectural and design models, and implementation in source code. The feature model is used as the starting point for cascaded refactorings for its

different categories correspond to other models. The set of models provide the foundation to specify and maintain traceability from the requirements to design and implementation of a framework.

- Propose a solution to identify and realize the variability of frameworks

Commonality and variability are identified and organized in the feature model. Their traceable information is specified by the trace maps between the set of models. Thus, traceability links of variability can be followed from the requirements, to design and implementation, in order to ensure and verify the realization of variability.

- Perform refactorings on a framework with a set of refactorings on the models

Refactoring of a framework is obtained through cascaded refactorings on the models. A trace map is used to discover the constraints and motivation of refactorings of range model from the refactorings of the domain model. Refactorings of different models are cascaded due to updating trace maps in order to maintain traceability. The “behaviour” of a framework is specified with different notions for each kind of models. The definition of refactorings of each model and the trace maps ensure the behaviour preservation of framework refactoring.

- Define a template to document refactoring decisions as part of the framework documentation

A decision record describes a refactoring of one model in terms of intent, preconditions, arguments, and impact. The intent specifies the motivation of the change; the arguments indicate the effected model elements; the preconditions capture the “snapshot” of the context before the refactoring; and the impact illustrates the consequence of the refactoring in the model, and is often considered as the preconditions of the consecutive refactoring. The content of decision records can be used as the rationale of framework refactoring, to aid the development and maintenance of framework, as part of the framework documentation.

Chapter 5

Conclusion

To accomplish great things, we must not only act,
but also dream; not only plan, but also believe.
~Anatole France

5.1 Overview

Software systems are becoming progressively more complex and expensive to build. Coding systems from scratch, with minimal leverage from one system to the next, clearly is not cost effective and is not a scalable means of construction [PRIE89]. Finding more economical ways of building software is a basic goal of software engineering. Software reuse is believed to be a key in achieving this goal [JGJ97].

Object-oriented application frameworks are a practical way to express reusable design across a domain [FSJ99]. A framework provides a generic design and the reusable implementation for a family of applications through a set of abstract classes and their collaborations [JF88]. The reuse is achieved through framework customization, which is typically done by subclassing the abstract classes and overriding a small number of methods.

Frameworks are more difficult to design and develop than individual applications [JOHN93] [SCHM97]. The framework design “must be simple enough to be learned, yet must provide enough features that it can be used quickly and enough hooks for features that are likely to change” [RJ97]. Existing framework development methodologies have suggested use domain analysis, software evolution, and design

patterns [FHLS99]. However, identifying the required variability for the family of applications and the designing mechanisms that provide the variability is a problem [BMMB00]. Furthermore, the evolution of the framework must be considered, especially as all frameworks seem to mature from initial versions through to a stable platform [SB99] [FSJ99].

The thesis introduces the cascaded refactoring methodology for framework development and evolution. The methodology is a hybrid approach that combines the modeling aspects of top-down domain engineering approaches, and the iterative refactoring process from the bottom-up approaches. It views framework development as framework evolution, which is achieved by framework refactoring followed by framework extension. The methodology focuses on framework refactoring, and extends the notion of refactoring that has been applied to source code, and to design in the form of class diagrams, to other models of frameworks. It specifies a framework with a set of models: the feature model, the use case model, the architectural model, the design model, and the source code; and relates the set of refactorings across those models through change impact analysis using the alignment maps. A *trace map* of two models specifies horizontal traceability links between the elements and relationships in the two models. An *alignment map* is a trace map that preserves commonality and variability. In our methodology, we assume that the trace maps used are indeed alignment maps, so the term trace map is used throughout. The framework refactoring is a set of refactorings of the models, and the constraints on how to refactor a particular model is determined or impacted by the previous refactorings via the trace maps between the models. Hence, the restructuring cascades from one model to the next. Overall, the refactoring of a framework is a set of model transformations that maps a coherent set of aligned models to another coherent set of aligned models. The methodology is validated by a case study, the **Know-It-All** framework for relational database management systems.

The remainder of this chapter is organized as follows. We will summarize the contributions and the limitations in section 5.2 and 5.3. The related work and validation issues are discussed in section 5.4 and 5.5. The future work is presented in the last section.

5.2 Contributions

Our work makes the following contributions:

1. **Cascaded refactoring is proposed as a methodology.**

As a common observation, framework design takes iteration [JF88]. The first version of a framework is usually a White-Box framework [RJ97]. Applications are built from the framework to validate its functionality, flexibility, etc. Thus, framework development is framework evolution, which can be viewed as framework refactoring followed by framework extension. A framework is specified with a set of models. The process of *cascaded refactoring* is a series of refactorings of the models. The impact of the refactorings on a model M_i to a model M_j , is translated via the trace maps that have M_i as the domain and M_j as the range. A refactoring on one model is not an isolated activity because the consequence of the refactoring can be the constraints to initiate refactorings on other models to preserve the traceability. A refactoring is always a part of the cascaded refactoring. Framework refactoring is achieved through a set of cascaded refactorings on the models of the framework.

2. **A set of models are chosen for the framework specification.**

Clear guidelines to document a framework with a set of coherent models are not given by the existing methodologies [FSJ99] [BMMB00]. We have carefully reviewed many software models and chosen the appropriate ones to specify a framework across the analysis, design, and implementation. Research in domain engineering has proved that the feature modeling is effective to identify and organize commonality and variability of a software product line [KCH+90] [KLL+02]. So, a feature model is chosen for the domain analysis. We select a use case model to capture the functionality, an architectural model to express the high level design in terms of layers and subsystems, a design model to specify the collaboration of classes and objects, and source code. Metamodels, as a precise definition of the constructs and rules needed to create models, are defined for the feature model and the architectural model. We also adopted Rui's use case metamodel [RB03]. Metamodels are defined to exactly specify the models, to clearly describe the refactorings and trace maps, and to allow the behaviour preservation of refactorings to be justified.

3. Trace maps between models are defined to aid traceability.

A framework has to not only encompass commonality of all applications that might be built from the framework, but also account for the variation that exists between those applications [FHLS99]. The existing methodologies have had little work on the identification and realization of the required variability of frameworks. The cascaded refactoring methodology stresses the model traceability to address this issue via alignment maps. A set of trace maps — actually alignment maps — are defined to specify and maintain the traceability links between the models. Commonality and variability of a framework are captured into the feature model. The consistency between the feature model and other models are maintained by using the trace maps. These maps cover the analysis, design, and implementation of the framework. The realization of the required variability is managed by mapping the requirements to the design and implementation. Furthermore, the impact of changes on the design and implementation due to the alteration on requirements of the framework is also handled by the trace maps. They deliver a solid foundation for the cascaded refactorings.

4. Partial set of refactorings on different models are defined.

Refactoring is a process of changing a software system in such a way that it does not alter the visible behaviour of the code, yet improves its internal structure [FOWL99]. Refactoring research has been mainly focused on source code and design level refactoring [OPDY92] [RBJ97] [TB99] [ROBE99] [MT04]. Our methodology extends the notion of refactoring to the feature model, the use case model, and the architecture. The invariants of refactoring of those models are clarified. A partial set of refactorings is defined for each of those models. The refactorings have been validated in the **Know-It-All** case study. The set of refactorings is by no means a complete list but the refactorings are sufficient for the case study.

5. A document template is defined to record the refactoring of models.

The cascaded refactoring methodology incorporates the issue-driven approach and views refactoring as an issue-driven activity, to maintain the traceability of the models via the trace maps. The choices of appropriate refactorings are based

upon the combination of constraints from the previous model refactorings, and the issue addressed by the present model refactoring. The preconditions of a refactoring must be satisfied to perform the refactoring. The overall refactoring rationale is a collection of decisions. Each decision records the intent, choice, arguments, and the consequences of a refactoring. Since a framework is always refactored until it reaches a mature platform, the refactoring document as part of the framework documentation benefits to the design and maintenance of the framework.

6. **An academic setting framework for relational database management systems, called Know-It-All, is developed as the case study to validate the methodology.**

Know-It-All provides a generic infrastructure for relational database management systems. The methodology has been validated with the development of Know-It-All. The trace maps of the Know-It-All models are specified and validated. Cascaded refactoring examples are demonstrated. The case study explores and validates key issues of the methodology: the notion of behaviour for the models, the notion of architectural model for framework architecture, the impact on traceability preservation by the trace maps, and the document template to record refactorings. The initial work focuses on refactorings of the feature model, use case model, and architectural model, since refactoring of source code and design have already been addressed by the refactoring community. Our publications have validated the refactoring concepts of the feature model, use case model, and architectural model [BX01] [BCC+02] [RB03].

5.3 Limitations

Weak points exist in our work due to the limited time and resources. They are summarized as follows:

1. **Cascaded refactoring approach is not a complete methodology.**

The methodology aims to cover framework evolution. However, our work only focuses on framework refactoring at current stage. Further study on framework

extension remains to be done. The metamodels are not defined in formal languages. We only have initial treatment of variability within framework models. The methodology lacks a thorough definition for variability in each of the models. The variability mappings are still not clearly specified. The trace maps are not full maps and are only defined on the subset of each model. Source code related trace maps are not defined. The invariants of different models during the cascaded refactorings are mainly concerned with the functionality. More quality attributes should be considered in the context of a framework. We have only defined a small subset of refactorings for the models. More refactorings are expected and the formal justification of behaviour-preservation for each refactoring is expected. The methodology only deals with White-Box level frameworks. It is possible that different documents and models are needed for other maturity level frameworks. However, we cannot draw any conclusion on this point at the current stage.

2. The architectural modeling is rather limited.

We had difficulty to choose an appropriate architectural model for a framework. Even UML does not provide specific mechanisms for architecture modeling. We took the traditional way of subsystem and interfaces to present the high-level framework architecture. Other views of architecture such as the process, and deployment views are not covered in the metamodel. The architectural model does not provide specific notion to represent hot spots, which are often used in framework design. The invariant of refactoring of architecture only takes the functionality into consideration. We have not decided which quality attributes should be preserved by framework architecture transformations. Our guess is to have a set of quality-preserving refactorings for each quality attribute.

3. The case study is not big enough.

Know-It-All only supports a small set of DBMS features. There are not many hot spots available to support enough variability, such as different data models, query languages, etc. The indexing techniques are quite limited. More work is needed on the physical layer. We only did a small number of refactorings on the framework. The framework has not been validated with the reuse “rule of thumb”, i.e. a framework should be verified a number of applications from the

framework in different contexts [RJ97].

5.4 Related Work

The work most closely related to the content of the thesis is the work on framework development and refactoring.

5.4.1 Framework Development

The existing framework development methodologies can be classified into Bottom-Up, Top-Down, Hot Spot Generalization, and Use Case Driven approaches [FSJ99].

The classic bottom-up approaches suggest an incremental, iterative way to build a framework [JF88] [JOHN93] [RJ97]. The framework development begins with developing several applications within different contexts in the framework domain. These applications are generalized to construct the White-Box framework. The framework is validated by re-developing those applications with the framework. More applications are developed and the framework is refactored to accommodate the necessary changes in order to handle the new applications. The iteration continues until all applications within the framework domain can be instantiated from the framework [WW93]. The bottom-up approaches do not specify the way in which the domain knowledge is obtained [FSJ99]. It is difficult and expensive to design the framework architecture by generalizing existing applications [JOHN93]. The approaches also emphasize refactorings in framework development but do not address the traceability issue between the different framework artefacts during refactorings [MENS05].

Top-Down approaches start with domain analysis to organize commonality and variability within the framework domain into analysis models [CHW98] [CN02]. The analysis result is used to design the Domain Specific Software Architecture (DSSA) and appropriate reusable components that can be instantiated during the application development. Organizational Domain Modeling (ODM) [STAR96] is a detailed domain analysis process with a set of work products and dossiers. Feature-Oriented Reuse Method (FORM) [KKL+98] proposes a layered feature model to categorize features and derives the architecture and class models with a general guideline. Lucient [WL99] proposes the FAST (Family-Oriented Abstraction, Specification, and Translation) methodology for product lines. FAST promotes very small product lines

which are well understood, so development is a one-increment activity. Software Engineering Institute (SEI) [CN00] initiates a program that integrates various practices into a framework for Product Line Practice. As Robert and Johnson [RJ97] point out, designing a generic architecture based upon the primary abstraction of the application requirements is very difficult. Furthermore, the top-down approaches do not provide adequate support to framework evolution since the approaches rely on a rather stable DSSA [WL99]. However, empirical studies have shown that software developers can only identify and predict a subset of future changes [LS98]. Coplien et al. [CHW98] also recognize that changes on a DSSA cannot be avoided due to the emergence of new applications with novel requirements. It is more practical to have an initial version framework based upon the partial domain analysis, then evolve the framework to meet new requirements.

Hot spot generalization approaches start with an object model which is intended to meet the domain-specific requirements of all applications of the framework domain [PG94] [SCHM97] [PREE99]. The object model is usually developed by domain experts and experienced software engineers with CRC cards [BC89] or use cases and scenarios [PREE99]. An iteration process is performed to identify the hotspots with variability classification, and to associate each one with a hot spot subsystem, which is usually realized by design patterns [GOF94]. Demeyer et al. [DRMG99] suggest a variation to the approaches that introduces a separate abstract class for each dimension of variability of a hot spot, i.e. put the hook methods called by a template method into different classes to increase flexibility. Hot spot generalization approaches rely on “perfect” domain experts since it is very challenging to design a good domain specific object model. They do not integrate domain analysis techniques and have no precise guidelines to organize the requirements in terms of commonality and variability into object models [FHLS99]. Moreover, the traceability from requirements to the design is not well addressed [HIM01].

Use case driven approaches plan all applications of the framework domain at once, and capture the requirements of those applications within use cases [JCJO92] [JGJ97]. The commonality and variability are specified with generalization and variation points in use cases. After the analysis, the approaches may then proceed with either the hot spot generalization or top-down approaches. RSEB [JGJ97] encompasses separated processes for Domain Engineering and Application Engineering. The approach

integrates the “4+1 View” [KRUC95] to model architectures. RSEB places an emphasis on modeling variability and keeping the traceability links from the requirements to design and implementation models. However, empirical studies have shown that use case models are sometimes not appropriate to model variability in complex systems [GFA98] [VAM+98]. Thus, Griss et al. [GFA98] extends RSEB to FeaturSEB by incorporating feature modeling as the domain analysis technique. The analysis model in the Catalysis approach [DW98] is based on three modeling concepts: type, the external behaviour of an object; collaboration, the interaction between objects; and refinement, the abstraction process to generalize type and collaborations. Architecture is developed and composed from the abstract concepts in the analysis model. Use case assortment [MMM99] combines a set of modeling heuristics with an analysis technique that identifies abstractions in the use cases from the application viewpoint, to construct abstract use cases and actors for framework development. The use case driven approaches are intended to use a set of models to specify the reusable architecture and components, and precise guidelines to map the requirements to design and implementation [JGJ97]. However, they have not precisely defined the traceability links between the entities and relationships of different models. Furthermore, they do not address the issues of framework evolution [BMMB00].

5.4.2 Refactoring

Refactoring is a behaviour-preserving program transformation that automatically updates an application’s design and underlying source code [FOWL99]. Opdyke [OPDY92] introduces the term “refactoring” and defines the concept of preconditions as the enabling conditions for a refactoring. He also defines a set of low-level and high-level refactorings for C++ programs. Many of them were implemented by Roberts in his Smalltalk refactoring browser [ROBE99]. Roberts extends Opdyke’s definition of refactoring by adding postconditions, which are assertions that a program must satisfy after the refactoring is applied. The idea comes from the observation that refactorings are typically applied in sequences intended to set up preconditions for later refactorings. Tokuda [TB95] [TOKU99] implements the refactorings proposed by Opdyke for C++, and proposes additional refactorings to support design patterns as target states for software restructuring efforts. Martin Fowler [FOWL99] explains the principles and best practices of refactorings, and gives a comprehensive catalogue

of refactorings. Each refactoring is given a name, a short summary, a motivation describes why the refactoring should be done, a step-by-step description of how to apply the refactoring, and an example. Refactoring is a key practice in eXtreme Programming (XP) [BECK99]. Software developers evolve the design incrementally upon new requirements from the clients. Two key aspects of XP are continual refactoring of the source code and unit testing for the justification of behaviour-preservation. Pipka [PIPK02] suggests only use behaviour-oriented testing since tests relying on the program structure may show different results due to the alteration of the structure by the refactorings.

Refactoring can be represented as graph transformations [HECK95]. A software artefact is represented as a graph, refactorings as graph production rules, and the application of a refactoring as a graph transformation. The research area of object-oriented software refactoring originates in the research on how to restructure object-oriented database schemas [BKKK87], since the schemas can be seen as the predecessor of the UML class diagrams. The approach is adopted by Opdyke [OPDY92] into object-oriented program refactoring. Heckel [HECK95] uses graph transformations to formally prove that any set of refactoring postconditions can be translated into an equivalent set of preconditions. Mens et al. [MDJ02] present the formalisation of refactoring using graph rewriting, a transformation that takes an initial graph as input and transforms it into a result graph. The preserved “behaviour” emphasizes on the implementation of each method that is involved in the refactoring. Philipps and Rumpe [PR97] propose a calculus for stepwise refinement of abstract data flow architecture style in terms of components and connectors. The calculus is composed of a set of graph transformation rules, which are justified by the refinement relations on a black-box view of the architecture. The Design Maintenance System (DMS) [BAXT92] is a rule-based transformation system that is applicable with a hierarchy of domains, each of which is specified by syntax, semantics, and mappings to the same or other domains. DMS can implement source code transformations such as COBOL programs. However, DMS transformations do not guarantee behaviour-preservation. It is feasible within DMS to define a domain corresponding to each of our models, to define a set of transformations for the models, and to define the trace maps between models. Thus, cascaded refactoring could be realized within DMS. Mens and D’Hondt [MD00] introduce an evolution contract formalism to manage

UML model transformation. The UML metamodel is extended to incorporate the concept of evolution contract, which is used to specify model transformations and to justify the behaviour-preservation. They claim that the formalism can handle the evolution of all kinds of UML models since it is defined at the metamodel level. It is sensible to use graph transformation as the formalism of refactoring because graphs are a language-independent representation of implementation. Moreover, transformation rules can be precisely defined, and the formalism allows the justification of behaviour-preservation [HECK95]. On the other hand, it is extremely complicated to deal with large nested structure with graph transformations, and the behaviour preservation still cannot be guaranteed [MT04].

While many techniques are available for program refactoring, some researchers shift their focus to other software artefacts. Steyaert et al. [SLMD96] propose the concept of reuse contract to handle change propagation between inheritance class hierarchy during software evolution. Evolution is specified with reuse operators, which define the transformation rules on class hierarchies. Mens [MENS01] extends their work to allow arbitrarily complex reuse contracts, in order to handle UML collaboration with graph rewriting rules. Judson et al. [JCF03] introduce a pattern-based metamodeling approach to evolve the UML design models by incorporating appropriate design patterns [GOF94] into the models. Transformations are described with transformation patterns, each of which specifies the created and deleted model elements by the transformation, and the preserved relationships between the target and source model elements. Garg et al. [GCC+03] present the M  nage graphical environment to manage the evolution of software product line architectures. M  nage uses a XML-based Architecture Description Language (xADL2.0) [WH02] to describe the architectures in terms of components, connectors, and their interfaces with schemas. Optional and variation points are expressed with Boolean expressions. Critchlow et al. [CDCH03] enhance the M  nage environment to automate a set of predefined architecture refactorings with two metrics tools. Their work emphasizes configuration management and automation support of the evolution of product line architectures. However, they neither specify the invariant of the architecture refactorings, nor the justification of behaviour-preservation. Back [BACK02] introduces an incremental refinement approach to evolve software architecture, which is viewed as a hierarchy

of layers. Behaviour preservation is justified with correctness conditions in the refinement calculus. However, the restructuring still focuses on the class level with the invariants of the attributes and methods of classes. Russo et al. [RB98] recommend to restructure natural language requirement specifications by decomposing them into a structure of viewpoints, each of which represents partial requirements of system components. They claim that refactorings increase the comprehension of requirements and detect inconsistencies.

Little work has been done on the evolution techniques of software artefacts other than source code [MENS05], and the traceability management between different artefacts during refactoring [MD03]. The cascaded refactoring methodology addresses the two issues with the extended refactoring concept and trace maps.

5.5 Validation Issue

The first part describes the validation of three seminal work on program refactoring in academic settings. The second part discusses how to validate the cascaded refactoring methodology in an ideal industry environment.

5.5.1 Validation in Academic Refactoring Community

Opdyke [OPDY92] identifies a set of C++ program restructurings from the survey of related work in his Ph.D. thesis. He introduces the term "refactoring" and suggests using pre-conditions to preserve the program behaviour during refactorings. He defines 23 low-level primitive refactorings and 3 composite refactorings which are composed by the primitive refactorings. For each of the low-level refactorings, Opdyke elaborates its preconditions and explains why it is behaviour preserving. He defines seven program properties as the invariants that the refactorings must hold. He also proposes three high-level complex refactorings and discusses their behaviour-preservation. He uses two simple C++ program segments to demonstrate how the refactorings work, and how those refactorings improve the quality of design and code. Opdyke also discusses the tool support for refactoring and mentions his research prototype that translates C++ source programs to Lisp forms for refactoring. However, he does not give any detail of the prototype nor explain how to use the prototype to validate his refactorings. He also admits that the practical utility of his work is not known for

the refactorings have not been validated in large program settings.

Roberts [ROBE99] takes the same view with Opdyke in that a high-level refactoring will be correct if the low-level refactorings which compose of the high-level refactoring are implemented correctly. However, he argues that it is often not practical to prove the exact “behaviour-preservation” due to the fact that many quality attributes can be interpreted as the “behaviour”. He defines several common refactorings by adding post-conditions into the refactoring definition, which is specified in the first order predicate logic. He claims that the definition of post-conditions allows the elimination of program analysis that is required within a chain of refactorings. He also proposes the dependency concept between refactorings based on commutativity and an approach to calculate the conditions under which any two refactorings may commute. He defines a method of calculating the preconditions for composite refactorings. Roberts [RBJ97] and his colleague have developed a Smalltalk refactoring tool, called the Refactoring Browser, to validate the refactoring ideas he proposed. He argues that the tool implements a subset of Opdyke’s primitive refactorings with run-time analysis and has been used by a number of software developers for refactoring work. He uses a simple refactoring to illustrate the design and usage of the tool.

Tokuda [TOKU99] views a refactoring as a parameterized behaviour-preserving program transformation. He implements Opdyke’s refactorings in C++, and contributes more refactorings which are defined by himself. He does not use post-conditions, and names “pre-condition” as “enabling condition” in refactoring definitions. The behaviour-preservation of each of the refactorings is discussed informally. He identifies three complex refactorings for object oriented design evolution. Tokuda uses a small C++ program as the example to illustrate how to evolve design with refactorings. He also presents experiment results of using refactorings to replicate the design evolution of two non-trivial C++ applications, by incorporating the Saga++ toolkit [BODI94]. The toolkit provides a semantic analyzer and an object-oriented programmer’s interface for modifying programs. Tokuda demonstrates the evolution of the applications with two refactoring examples. He proposes additional program invariants because his experiments and analysis show that the invariants proposed by Opdyke are not sufficient to preserve behaviour.

5.5.2 Ideal Industry Validation

The ideal industrial evaluation would occur in a software organisation that already had several frameworks in existence. These were being actively applied to produce applications and undergoing evolution as a consequence. It would be even better if the frameworks were at different levels of maturity.

It is essential that the software organisation have a system for measurement in place that was actively used. The basic tracking of test results, bug reports (i.e. defect tracking) could provide quantitative information about the quality of the framework. Measurement of the effort required for developing an application is necessary to evaluate productivity and productivity changes. These measurements must be normalized somehow by reliable estimates of application size.

The system for measurement must have been in place for several years so that a reliable benchmark (norm) has been established for the existing practices of application development and framework evolution. The effect of the cascaded refactoring methodology on improvements in productivity or framework quality could be evaluated against this norm.

To our best knowledge, there is no consensus on an effective and accurate metric system for framework measurements [FSJ99] [BOSC02]. Alshayeb and Li [AL03] have conducted an empirical study and claimed that the existing object oriented metrics are considerably ineffective in the framework context due to the long cycled framework evolution process as the nature of the development. Bosch [BOSC98] argues that it is extremely difficult to estimate the development cost of application frameworks because of the abstract structure, high-level variability, and continuous iterative process in framework development. He also observes the fact that estimation techniques typically do not consider variability in general. Boehm et al. [BBMY04] propose a software product line life cycle economics model, the Constructive Product Line Investment Model (COPLIMO) that focuses on the implementation level in terms of line of code, which does not fit well in our context. Zubrow and Chastek [ZC03] define a small set of measures to estimate the development of software product lines. We would suggest three metrics from their set since the set are recommended by the Software Engineering Institute (SEI) Software Product Line Initiative [CN00].

1. *Total framework development cost*: It is composed of the cost of initial framework development, and the cost of evolution and maintenance. The measurement unit is man-hour.
2. *Customer satisfaction*: The data is collected by doing survey with the customers about framework quality, including the realization of required variability, flexibility, and the productivity of building applications from the framework.
3. *Market feature coverage*: The metric captures the extent to which the features are available in the framework cover those related to the target market in terms of percentage.

A pilot study of the cascaded refactoring methodology would involve at least one framework over the course of 3 to 5 iterations of applications development and the associated evolution of the framework. The measurement system would track productivity and framework quality. Each iteration would take about 6 weeks.

Before the pilot study could commence, there would need to be tool support for the cascaded refactoring methodology. Three kinds of tools are desirable: modeling tools, which are capable of creating and editing the models in the methodology; mapping tools, which facilitate the definition of the alignment maps and their use in the tracing dependencies both horizontally and vertically through the models; and refactoring tools, such as plug-ins for the Eclipse Integrated Development Environment (IDE) [DFK04] which allow the refactorings of each of the models and have a catalogue of refactorings implemented already.

An initial period of maybe 10 to 20 weeks would be required for the development team to construct the set of aligned models for the framework that the cascaded refactoring methodology requires. The development team would already be familiar with the framework and its domain. Hence, the overall time for an industrial trial would be approximately one year and involve a small number of experienced developers.

This would be a minimal industrial pilot study. It would be preferable to perform the trial with several frameworks at differing stages of maturity, and also to carry out more than the minimal number of iterations of application development and evolution per framework. Economic factors would probably rule this out in an industrial setting.

5.6 Future work

The methodology is still in its infancy stage. We want to have a set of solid metamodel for each model used in the methodology. The semantics of the metamodels should be verified with formal reasoning processes. A specific refactoring rule for each concept in every metamodel has to be defined. More refactorings should be defined for the feature model, the use case model, and the architectural model. The trace maps should cover the full set of the models. We still desire a consistent, coherent, aligned set of models and maps.

We need a better architectural modeling technique. Other views should be included in the architecture. We may be able to incorporate the whole Siemens approach into the architectural model.

A subproject to develop modeling and refactoring tools for the use case models is underway, and we hope to initiate a similar subproject for the architectural models soon. These should shed much light on the gaps in the methodology.

The **Know-It-All** framework case study has completed a partial domain analysis and prototyped its architectural design in C++ for the relational database management systems. The query optimization subframework has been completed, resulting in an implementation, models, and documentation on its use. The Gist framework for index techniques is going to be completed soon. Potential future work on **Know-It-All** also includes a subframework for physical storage, a subframework for hash indexes, and a subframework for inverted file indexes. The range of data models included in the framework will also be extended.

We expect more applications populated with the customization of **Know-It-All**. This will not only validate the framework itself, but also can testify the completeness and correctness of the cascaded refactoring approach. GraphLog [CEH+94] is a graph query language extending Datalog and negation. The language has recursion, usually as transitive closure, and has path expressions, which are similar to regular expressions. The GraphLog interface, as a standalone system exists [BWWZ05], and over the longer term both Coral [RSSS94] and GraphLog will be incorporated as data models in the **Know-It-All** framework.

Bibliography

- [AB97] M. Andersson and J. Bergstrand. *Formalizing Use Cases with Message Sequence Charts*. Master thesis, Department of Communication Systems at Lund Institute of Technology, 1997. Available at: <http://www.efd.lth.se/~d87man/EXJOBB/PS/ExBook.ps.Z.uu>
- [ACKE96] P. Ackermann. *Developing Object-Oriented Multimedia Software - Based on the MET Application Framework*. dpunkt Verlag, Heidelberg, Germany, 1996.
- [AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language*. Oxford University Press, New York, 1977. ISBN: 0195019199
- [AL03] M. Alshayeb, W. Li. *An empirical validation of object-oriented metrics in two different iterative software processes*. IEEE Transactions on Software Engineering, 29, 11, 1043 - 1049, November 2003.
- [ARAN94] G. Arango. *Domain Analysis Methods*. In Masao Matsumoto, Ruben Prieto-Diaz, Wilhelm Schafer. *Software Reusability*. Prentice Hall, UK, 1994. ISBN: 0130639184
- [ASTE02] D. Astels. *Refactoring with UML*. In Proceedings of 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering, 67-70, Alghero, Sardinia, Italy, 2002.
- [BACK02] R. J. Back. *Software construction by stepwise feature introduction*. In Proceedings of 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B, 162-183, Grenoble, France, January 2002.
- [BANS00] J. Bansiya. *Evaluating framework architecture structural stability*. ACM Computing Surveys, 32, 1, No. 18, March 2000.

- [BAXT92] I. Baxter. *Design Maintenance Systems*. Communications of the ACM, 35,4, 73-89,1992.
- [BBG+89] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. *Genesis: an extensible database management system*. IEEE Transactions on Software Engineering 14, 11, 500-518, 1989.
- [BBG+00] G. Butler, E. Bornberg-Bauer, G. Grahne, F. Kurfess, C. Lam, J. Paquet, I. Rojas, R. Shinghal, L. Tao, A. Tsang. *The BioIT projects: Internet, database and software technology applied to bioinformatics*. In SS-GRR'2000, Suola Superiore, G. Reiss Romoli SpA, Coppoto, Italy, July 2000. At: <http://www.ssgrr.it/en/ssgrr2000/proceedings.htm>
- [BBMY04] B.Boehm, A. Brown, R. Madachy, Y. Yang. *A software product line life cycle cost estimation model*. In Proceedings of 2004 International Symposium on Empirical Software Engineering (ISESE'04), 156-164, Los Angeles, USA, Aug. 2004.
- [BC87] K. Beck, W. Cunningham. *Using pattern languages for object-oriented programs* Position Paper for the Specification and Design for Object-Oriented Programming Workshop, 3rd Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Orlando, Florida, USA, October 1987.
- [BC89] K. Beck, W. Cunningham. *A laboratory for teaching object oriented thinking*. In Proceedings of Conference on Object Oriented Programming Systems Languages and Applications(OOPSLA'89), 1-6, New Orleans, Louisiana, USA, October 1989.
- [BCC+02] G. Butler, L. Chen, X. Chen, A. Gaffar, J. Li, L. Xu. *The Know-It-All Project: A Case Study in Framework Development and Evolution*, Domain Oriented Systems Development: Perspectives and Practices. K. Itoh, S. Kumagai, T. Hirota (eds), 101-118, Taylor&Francis, UK, 2002. ISBN: 0415304504
- [BCK97] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 1997. ISBN: 0201199300

- [BCKR97] G. Butler, A. Cretu, F. Khendek. *Reconciling Use Cases and Operational Profiles*. 1997. At: <http://citeseer.ist.psu.edu/148512.html>
- [BD99] G. Butler and P. Dénommée. *Documenting frameworks*, Building Application Frameworks: Object-Oriented Foundations of Framework Design. M. E. Fayad, D. C. Schmidt, R. E. Johnson (eds), 495-504, John Wiley & Sons, NY, 1999. ISBN: 0471248754
- [BECK99] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999. ISBN: 0201616416
- [BERA93] E. Berard. *Essays on Object-Oriented Software Engineering*. Prentice Hall, UK, 1993. ISBN: 0132888955
- [BGK98] G. Butler, P. Grogono and F. Khendek, *A reuse case perspective on documenting frameworks*. In Proceedings of the 5th Asia-Pacific Software Engineering Conference, 94-101, Taiwan, December 1998.
- [BJ94] K. Beck, and R. Johnson. *Patterns generate architectures*. In Proceedings of ECOOP'94, 139-149, Springer-Verlag, Berlin, German, 1994.
- [BK96] D. Binkley and K. Gallagher. *Program slicing*. Advances of Computing, 43, 1-50, 1996.
- [BKKK87] J. Banerjee, W. Kim, H. Kim, H.F. Korth. *Semantics and implementation of schema evolution in object-oriented databases*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 311-322, 1987.
- [BL76] R. A. Belady and M. M. Lehman. *A model of large program development*. IBM Systems Journal 15,1, 225-252, 1976.
- [BMMB00] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson. *Object-oriented framework-based software development: problems and experiences*. ACM Computing Surveys, 32, 1, No.3, March 2000.
- [BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, NY, 1996. ISBN: 0471958697

- [BODI94] F. Bodin. *Sage++: an object-oriented toolkit and class library for building fortran and C++ restructuring tools*. In Proceedings of the 2nd Object-Oriented Numerics Conference, Sunriver, Oregon, April 1994.
- [BOEH88] B.W. Boehm. *A spiral model of software development and enhancement*. IEEE Computer, 21,5, 61-72, May 1988.
- [BOOC94] G. Booch. *Designing an application framework*. Dr. Dobb's Journal 19, 2, 24-32, February 1994.
- [BOSC98] J. Bosch. *Product-line architectures in industry: a case study*. In Proceedings of the 21st International Conference on Software Engineering, Los Angeles, California, USA, 544-554, November 1998.
- [BOSC00] J. Bosch. *Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley, Reading, MA, 2000. ISBN: 0201674947
- [BOSC02] J. Bosch. *Maturity and evolution in software product lines: approaches, artefacts and organization*. In Proceedings of the 2nd International Conference on Software Product Line (SPL02), 257-271, San Diego, CA, August 2002.
- [BP98] M. Blaha, W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice-Hall, NJ, 1998. ISBN: 0131238299
- [BRJ99] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999. ISBN: 0201571684
- [BRYA94] A. D. Bryant. *Creating Successful Bulletin Board Systems*. Addison-Wesley, Reading, MA, 1994. ISBN: 0201626683
- [BSF02] P. Boger, T. Sturm, P. Fragemann. *Refactoring browser for UML*. In Proceedings of 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP2002), 77-81, Sardinia, Italy, May 2002.
- [BSX03] G. Butler, X. Shen, L. Xu. *Issues in architectural modeling and evolution in the Know-It-All case study*. In Proceedings of 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'03), 321-330, Huntsville, AL, USA, April 2003.

- [BUHR98] R. Buhr. *Use case maps as architectural entities for complex systems*. IEEE Transactions on Software Engineering 24, 12, 1131-1155, 1998.
- [BUTB99] G. Butler. *Database technology for pathways*. In Workshop on Computation of Biochemical Pathways and Genetic Networks, E. Bornberg-Bauer, A. deBeuckelaer, U. Kummer, U. Rost (eds), 89-95, Logos Verlag, Berlin, 1999. ISBN: 3897220938
- [BUTL99] G. Butler. *Developing frameworks by aligning requirements, design, and code*. In Proceedings of 9th Workshop on Software Reuse (WISR-9), Austin, Texas, January 1999.
- [BUTL02] G. Butler. *Architectural refactoring in framework evolution: A case study*, Generative Programming and Component Engineering, LNCS 2487, 128-139, ACM Press, 2002.
- [BWZ05] G. Butler, G. Wang, Y. Wang, L. Zou. *A graph database with visual queries for genomics*. In Proceedings of 3rd Asia-Pacific Bioinformatics Conference (APBC'05), 31-40, Singapore, January 2005.
- [BX01] G. Butler and L. Xu. *Cascaded refactoring for framework evolution*. ACM SIGSOFT Software Engineering Notes, Proceedings of 2001 Symposium on Software Reusability 26,3, 51-57. 2001.
- [CDCH03] M. Critchlow, K. Dodd, J. Chou, A. V. Hoek. *Refactoring product line architectures*. In Proceedings of 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE03). Victoria, BC, Canada, November 2003. At: <http://www.ics.uci.edu/simandre/research/papers/REFACE2003.pdf>
- [CE00] K. Czarnecki, U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, Reading, MA, 2000. ISBN: 0201309777
- [CEH+94] M.P. Consens, F.C. Eigler, M.Z. Hasan, A.O. Mendelzon, E.G. Noik, A.G. Ryman, and D. Vista. *Architecture and applications of the Hy+ visualization system*. IBM Systems Journal 33,3, 458-476, 1994.

- [CHOT99] S. Clarke, W. Harrison, H. Ossher, P. Tarr. *Subject-oriented design: towards improved alignment of requirements, design and code*. In Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Denver, Colorado, U.S., November 1999.
- [CHW98] J. Coplien, D. Hoffman, D. Weiss. *Commonality and variability in software engineering*. IEEE Software, 15,6, 37-45, November 1998.
- [CJH00] S. Cook, H. Ji, R. Harrison. *Software evolution and software evolvability*. Unpublished manuscript, University of Reading, UK, 2000.
- [CN00] P. Clements, L. Northrop. *A framework for software product line practice*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh. The Product Line Practice (PLP) Initiative. At: <http://www.sei.cmu.edu/plp/2000>.
- [CN02] P. Clements, L. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley, Reading, MA, 2002. ISBN:0201703327
- [COCK97] A. Cockburn. *Structuring use cases with goals*. Journal of Object-Oriented Programming, ROAD, 10,5, 35-40 and 10,7,56-62,1997.
- [Codd70] E. F. Codd. *A relational model of data for large shared data banks*. Communications of the ACM 13,6,377-387, June 1970.
- [CONK89] J. Conklin. *Design rationale and maintainability*. In Proceedings of 22nd Annual Hawaii International Conference on System Science, 2, 533-539. Hawaii, January 1989.
- [CP95] S. Cotter and M. Potel. *Inside Taligent Technology*. Addison-Wesley, Reading, MA, 1995. ISBN: 0201409704
- [CHSV97] W. Codenie, K. Hondt, P. Steyaert, A. Vercammen. *From custom applications to domain-specific frameworks*. Communications of the ACM, 40, 10, 71-77, October 1997.
- [DESA90] B.C. Desai. *An Introduction to Database Systems*. West Publishing Company, St. Paul, MN, 1990. ISBN:0314667717

- [DEUT89] L. P. Deutsch. *Design reuse and frameworks in the Smalltalk-80 system*. Software Reusability, 2, Applications and Experience, 57-71, 1989.
- [DFK04] J. D’Anjou, S. Fairbrother, D. Kehn. *The Java Developer’s Guide to Eclipse*. Addison-Wesley, Reading, MA, 2004. ISBN: 0321305027
- [DKO+97] D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson. *Applying software product-line architecture*. Computer 30,8, 49-55, August 1997.
- [DRMG99] S. Demeyer, M. Rieger, T. D. Meijler and E. Gelsema. *Class composition for specifying framework design*. Theory and Practice of Object Systems (TAPOS) 5,2,73-81, April 1999.
- [DW98] D. F. D’Souza, A. C. Wills. *Objects, Components, and Frameworks with UML—The Catalysis Approach*. Addison Wesley, Reading, MA, 1998. ISBN: 0201310120
- [DW99] D. F. D’Souza, A. C. Wills. *Composing modeling frameworks in Catalysis*, Building Application Frameworks: Object-Oriented Foundations of Framework Design. M. E. Fayad, D. C. Schmidt, R. E. Johnson (eds), 441-460, John Wiley & Sons, NY, 1999. ISBN: 0471248754
- [EGYE01] A. Egyed. *A scenario-driven approach to traceability*. In Proceedings of 23rd International Conference on Software Engineering (ICSE’01), 123-132, Toronto, Canada, May 2001.
- [FHG98] D.G. Firesmith, B. Henderson-Sellers, I. Graham. *OPEN Modeling Language (OML) Reference Manual*. Cambridge University Press, NY, 1998. ISBN: 0521648238
- [FHLS97] G. Froehlich, H. J. Hoover, L. Liu and P. Sorenson. *Hooking into object-oriented application frameworks*. In Proceedings of 19th International Conference on Software Engineering, 491-501, 1997.
- [FHLS99] G. Froehlich, H. J. Hoover, L. Liu and P. Sorenson. *Designing object-oriented frameworks*, Handbook of object technology. S. Zamir(ed), 25-1-25-22, CRC Press LLC, FL, 1999. ISBN: 0849331358

- [FINK91] A. C. Finkelstein. *Tracing back from requirements*. IEEE Colloquium on Tools and Techniques for Maintaining Traceability During Design, 7/1 - 7/2, London, 1991.
- [FOWL99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999. ISBN: 0201485672
- [FREE83] P. Freeman, editors. *Reusable software engineering: concepts and research directions*. In Proceedings of the ITT Workshop on Reusability in Programming, IEEE Computer Society Press, 129-137, September 1983.
- [FSJ99] M. E. Fayad, D. C. Schmidt, and R. E. Johnson, editors. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons, NY, 1999. ISBN: 0471248754
- [GARL00] D. Garlan. *Software architecture: a roadmap*. In Proceedings of the Conference on The Future of Software Engineering, ACM Press, 91-101, 2000.
- [GBS01] J. V. Gorp, J. Bosch, M. Svahnberg. *On the notion of variability in software product lines*. In Proceedings of 1st Working IEEE Conference on the Software Architecture, (WICSA), IEEE Computer Society, 2001.
- [GCC+03] A. Garg, M. Critchlow, P. Chen, C. V. Westhuizen, A. V. Hoek. *An environment for managing evolving product line architectures*. In Proceedings of 19th International Conference on Software Maintenance, (ICSM'03), 358-368, Amsterdam, Netherlands. September 2003.
- [GF94] O. C. Gotel and A. C. Finkelstein. *An analysis of the requirements traceability problem*. In Proceedings of 1st International Conference on Requirements Engineering, (ICRE'94), 94-101, Colorado Springs, Colorado, USA, 1994.
- [GFA98] M. L. Griss, J. Favaro, and M. d'Alessandro. *Integrating feature modeling with the RSEB*. In Proceedings of 5th International Conference on Software Reuse, 76-85, IEEE Computer Society, 1998.
- [GHM98] J. Grundy, J. Hosking, W. B. Mugridge. *Inconsistency management for multiple-View software development environments*. IEEE Transactions on Software Engineering, 24, 11, 960-981, November 1998.

- [GIBS97] J. P. Gibson. *Feature requirements models: understanding interactions*. In Feature Interactions In Telecommunications IV, 46-60, IOS Press, June 1997.
- [GJKT97] H. Gall, M. Jazayeri, R. R. Klösch, G. Trausmuth. *Software evolution observations based on product release history*. In Proceedings of 13th International Conference on Software Maintenance (ICSM'97), 160-168, Bari, Italy, October 1997.
- [GOF94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994. ISBN: 0201633612
- [GRAE96] G. Graefe. *Iterators, schedulers, and distributed-memory parallelism*. Software - Practice and Experience 26,4,427-452, 1996.
- [GRIS91] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.
- [HE90] B. Henderson-Sellers, J.M. Edwards. *Object-oriented software system life cycle*. Communication of ACM 33,9, 142-159,1990.
- [HECK95] R. Heckel. *Algebraic Graph Transformations with Application Conditions*. Master thesis, TU Berlin, 1995.
- [HF97] B. Henderson-Sellers, D. G. Firesmith. *Choosing between UML and OPEN*. American Programmer 10,3,15-23, 1997.
- [HFG97] B. Henderson-Sellers, D.G. Firesmith, I. Graham. *OML metamodel: relationships and state modeling*. Journal of Object Oriented Programming 10,1,March/April, 1997.
- [HIM01] T. Hayase, N. Ikeda, K. Matsumoto. *A three-view model for developing object-oriented frameworks*. In Proceedings of 39th International Conference And Exhibition On Technology Of Object-oriented Languages And Systems, 108-119, Santa Barbara, California, July 2001.

- [HKP97] J. M. Hellerstein, E. Koutsoupas, and C. H. Papadimitriou. *Towards an analysis of indexing schemes*. In Proceedings of 16th ACM SIGACT-SIGMOD-SIGART Symposiums on Principles of Database Systems, 249-256, Tucson, AZ, May 1997.
- [HNP95] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. *Generalized search trees for database systems*, In Proceedings of International Conference on Very Large Data Bases, VLDB'1995, 562-573, Zurich, Switzerland, September 1995.
- [HNS99] C. Hofmeister, R. Nord, D. Soni. *Applied Software Architecture*. Addison-Wesley, Reading, MA, 1999. ISBN: 0201325713
- [HP94] Hewlett Packard, Matra Marconi Space, CAP Gemini Innovation. *Domain Analysis Method*. Deliverable D3.2B, PROTEUS ESPRIT project 6086, 1994.
- [IEEE93] IEEE. *Report on the IEEE STD 1219-1993-standard for software maintenance*. ACM SIGSOFT Software Engineering Notes, 18, 4, 94-95, 1993.
- [JACO87] I. Jacobson. *Object-oriented development in an industrial environment*. In Proceedings of Conference on Object Oriented Programming Systems Languages and Applications, (OOPSLA'87), 183191. Orlando, Florida, USA. October 1987.
- [JB02] M. Jaring and J. Bosch. *Representing variability in software product lines: a case study*. In Proceedings of SPLC 2002, 15-36, San Diego, CA, USA, August 2002.
- [JBR99] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Reading, MA, 1999. ISBN: 0201571692
- [JCF03] S. R. Judson, D. L. Carver, R. France. *A metamodeling approach to model refactoring*. At: http://www.cs.colostate.edu/~france/publications/Judson_UML2003.pdf
- [JCJO92] I. Jacobson, M. Christenson, P. Jonsson, G. Overgaard. *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison-Wesley, Reading, MA, 1992. ISBN: 0201544350

- [JEJ95] I. Jacobson, M. Ericsson, A. Jacobson. *The Object Advantage: Business Process Reengineering with Object Technology*. Addison-Wesley, Reading, MA, 1995. ISBN: 0201422891
- [JF88] R. E. Johnson and B. Foote. *Designing reusable classes*. Journal of Object-Oriented Programming, 2, 1, 22-35, July 1988.
- [JGJ97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, Reading, MA, 1997. ISBN: 0201924765
- [JOHN92] R. E. Johnson. *Documenting frameworks using patterns*. In Proceedings of Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'92), 63-76, Vancouver, Canada. October 1992.
- [JOHN93] R. E. Johnson. *How to design frameworks*. Tutorial Notes, OOPSLA'93, Washington, D.C. October 1993.
- [JOHN97] R. E. Johnson. *Frameworks = (components + patterns)*. Communications of the ACM, 40, 10, 39-42, October 1997.
- [JR91] R. E. Johnson, V. F. Russo. *Reusing Object-Oriented Design*. Technical Report UIUCDCS-R-91-1696, University of Illinois, 1991.
- [KARL95] E. A. Karlsson. *Software Reuse: A Holistic Approach*. John Wiley & Sons, NY, 1995. ISBN: 0471958190
- [KCH+90] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. 1990.
- [KD98] N. Kabra, D. J. DeWitt. *Efficient re-optimization of sub-optimal query execution plans*. In Proceedings of 1998 SIGMOD Conference, 106-117, Seattle, Washington, USA, June 1998.
- [KD99] N. Kabra and D. J. DeWitt. *OPT++: an object-oriented implementation for extensible database query optimization*. The VLDB Journal 8, 1, 55-78, January 1999.

- [KELL90] C. Kelley. *Does it fit the bill?*. International Journal of General Systems, 18, 6, 3234, 1990.
- [KEAN97] L. Kean. *Feature-Based design rationale capture method for requirements tracing*. Software Engineering Institute (SEI), Carnegie Mellon University. 1997. At: http://www.sei.cmu.edu/str/descriptions/featbased_body.html
- [KK03] P. Kroll, P. Kruchten. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison-Wesley, Reading, MA, 2003. ISBN: 0321166094
- [KKL+98] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. *FORM: a feature-oriented reuse method with domain-specific reference architectures*. Annals of Software Engineering, 5, 143-168, 1998.
- [KKLL99] K. C. Kang, S. Kim, J. Lee, K. Lee. *Feature-oriented engineering of PBX software for adaptability and reusability*. Software- Practice and Experience 29,10,875-896, 1999.
- [KLL+02] K.C. Kang, K. Lee, J. Lee, S. Kim. *Feature oriented product line software engineering: principles and guidelines*, Domain Oriented Systems Development: Perspectives and Practices. K. Itoh, S. Kumagai and T. Hirota (eds) 29-46, Taylor&Francis, UK, 2002. ISBN: 0415304504
- [KMH97] M. Kornacker, C. Mohan, and J. M. Hellerstein. *Concurrency and recovery in generalized search trees*. In Proceedings of ACM-SIGMOD International Conference on Management of Data, 62-72, Tucson, AZ, May 1997.
- [KNET02] A. von Knethen. *Change-oriented requirements traceability: support for evolution of embedded systems*. In Proceedings of 18th International Conference on Software Maintenance, (ICSM'02), 482-485, Montreal, Canada, October 2002.
- [KR87] D. Kafura and G. R. Reddy. *The use of software complexity metrics in software maintenance*. IEEE Transactions on Software Engineering, 13, 3, 335-343, March 1987.

- [KRAS88] G. E. Krasner and S. T. Pope. *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80*. Journal of Object-Oriented Programming, 1,3,26-49, 1988.
- [KRUC95] P. Kruchten. *The 4+1 view model of architecture*. IEEE Software 12,6,42-50, November 1995.
- [KS96] C. D. Klingler and J. Solderitsch. *DAGAR: A process for domain architecture definition and asset implementation*. In Proceedings of ACM Conference on TRI-Ada'96: disciplined software development with Ada, 231-245, Philadelphia, PA, December 1996.
- [KS00] J. Kuusela, J. Savolainen. *Requirements engineering for product families*. In Proceedings of 22nd International Conference on Software Engineering, 61-69, Limerick, Ireland, June 2000.
- [KWON03] J.H. Kwon. *A Feature Model of The Oracle 9i Database Server*. Master thesis, Concordia University, 2003.
- [LB85] M. M. Lehman, L. A. Belady. *Program Evolution: Processes of Software Change*. Academic Press, London, 1985. 0124424406
- [LEHM96] M. M. Lehman. *Laws of software evolution revisited*. In Proceedings of 5th European Workshop on Software Process Technology, 108-124, October 1996.
- [LH89] K. Lieberherr and I. Holland. *Assuring good style for object-oriented programs*. IEEE Software, 6, 5, 38-48, September 1989.
- [LHR88] K. J. Lieberherr, I. Holland, A. J. Riel. *Object-oriented programming: an objective sense of style*. SIGPLAN Notices, 11, 323-334, September 1988.
- [LI93] H. Li. *Reuse-in-the-large: modeling, specification and management*. Advances in Software Reuse, 2nd International Workshop on Software Reusability, 56-65, Lucca, Italy, March 1993.
- [LIM96] W.C. Lim. *Reuse economics: a comparison of seventeen models and directions for future research*. In Proceedings of 4th International Conference on Software Reuse, 41-50, April 1996.

- [LIND94] M. Lindvall. *A Study of Traceability in Object-Oriented Systems Development*. PhD thesis, Linköping University, 1994.
- [LKCC00] K. Lee, K. C. Kang, W. Chae, and B. W. Choi. *Feature-based approach to object-oriented engineering of applications for reuse*. *Software— Practice and Experience* 30, 9, 1025-1046, 2000.
- [LL01] T. C. Lethbridge and R. Laganriere. *Object-Oriented Software Engineering, Practical Software Development Using UML and Java*. The McGraw-Hill Education, UK, 2001. ISBN: 0072834951
- [LN97] D. B. Lange, Y. Nakamura. *Object-oriented program tracing and visualization*. *Computer* 30, 5, 63-70, May 1997.
- [LR98] M. M. Lehman and J. F. Ramil. *Implications of laws of software evolution on continuing successful use of COTS software*. Department of Computing Technical Report 98/8, Imperial College, London, June 1998.
- [LR01] M. M. Lehman and J. F. Ramil. *Evolution in software and related areas*. In *Proceedings of 4th International Workshop on Principles of Software Evolution*, 1-16, Vienna, Austria, September 2001.
- [LR02] M. M. Lehman and J. F. Ramil. *Software uncertainty*. 1st International Conference on Computing in an Imperfect World, 8-10, Belfast, North Ireland, April 2002.
- [LR03] M. M. Lehman and J. F. Ramil. *Software evolution: background, theory, practice*. *Information Processing Letters* 88, 1-2, 33-44, October 2003.
- [LS96] M. Lindvall and K. Sandahl. *Practical implications of traceability*. *Software Practice and Experience*, 26, 10, 1161-1180. 1996.
- [LS98] M. Lindvall and K. Sandahl. *How well do experienced software developers predict software change?*. *Journal of Systems and Software* 43, 1, 19-27, 1998.
- [MB96] K. U. Mätzel, W.R. Bischofberger. *The any framework: a pragmatic approach to flexibility*. In *Proceedings of 2nd USENIX Conference on Object-Oriented Technologies and Systems*, 179-190, Toronto, Canada, June 1996.

- [MB97] K. U. Mätzel, W.R. Bischofberger. *Designing object systems for evolution*. Theory and Practice of Object Systems 3,4,265-283, 1997.
- [MB99] M. Mattsson, J. Bosch. *Observations on the evolution of an industrial OO framework*. In Proceedings of IEEE International Conference on Software Maintenance (ICSM'99), 139-145, August 1999.
- [MBZR03] T. Mens, J. Buckley, M. Zenger, A. Rashid. *Towards a taxonomy of software evolution*. In Proceedings of 2nd Workshop on Unanticipated Software Evolution. Warsaw, Poland, April 2003. Available at: <http://www.cs.uni-bonn.de/~gk/use/2003/Papers/18500066.pdf>
- [MCRL89] P. Madany, R. Campbell, V. Russo, and D. Leyens. *A class hierarchy for building stream-oriented file systems*. In Proceedings of ECOOP'89, 311-328, Nottingham, UK, July 1989.
- [MD00] T. Mens, T. D'Hondt. *Automating support for software evolution in UML*. Automated Software Engineering, 7, 1, 39-59, March 2000.
- [MD03] T. Mens, A. V. Deursen. *Refactoring: emerging trends and open problems*. In Proceedings of 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE03). Victoria, BC, Canada, November 2003. At: <http://homepages.cwi.nl/~arie/papers/refactoring/reface03.pdf>
- [MDJ02] T. Mens, S. Demeyer, and D. Janssens. *Formalising behaviour preserving program transformations*. Graph Transformations, volume 2505 of Lecture Notes in Computer Science, 286-301, 2002.
- [MEJD04] T. Mens, N. V. Eetvelde, D. Janssens, S. Demeyer. *Formalising refactorings with graph transformations*. Journal of Software Maintenance and Evolution, (Submitted March 9, 2004). At: http://www.fots.ua.ac.be/graphtransfo_refactoring/FI-refactoring.pdf
- [MENS01] T. Mens. *A formal foundation for object-oriented software evolution*. In Proceedings of 17th International Conference on Software Maintenance (ICSM'01), 549-552, Florence, Italy, November 2001.

- [MENS05] T. Mens. *Challenges in software evolution*. Position paper in the joint ECRIM-ESF workshop ChaSE 2005, Berne, Switzerland, April, 2005.
- [MMM99] G. Miller, J. McGregor, and M. Major. *Capturing Framework Requirements*, Building Application Frameworks: Object-Oriented Foundations of Framework Design. M. E. Fayad, D. C. Schmidt, and R. E. Johnson (eds). 309-324, John Wiley & Sons, NY, 1999. ISBN: 0471248754
- [MMW98] L. M. Mackinnon, D.H. Marwick and M.H. Williams. *A model for query decomposition and answer construction in heterogeneous distributed database systems*. Journal of Intelligent Information Systems, 11, 69-87, 1998.
- [MRB97] R. C. Martin, D. Riehle, and F. Buschmann. *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, MA, 1997. ISBN: 0201310112
- [MT00] N. Medvidovic, R.N. Taylor. *A classification and comparison framework for software architecture description languages*. IEEE Transactions on Software Engineering, 26, 1, 70-93, 2000.
- [MT04] T. Mens and T. Tourwé. *A survey of software refactoring*. IEEE Transactions on Software Engineering, 30, 2, 126-139, 2004.
- [NIE03] B.NIE. *A Tree Index Framework For Databases*. Master thesis, Concordia University, 2003.
- [NR68] P. Naur, B. Randell. *Software engineering: report on a conference sponsored by the NATO science committee*. Garmisch, Germany, October 1968. At: <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>
- [OG02] T. Olsson and J. Grundy. *Supporting traceability and inconsistency management between software artifacts*. In Proceedings of 6th IASTED International Conference on Software Engineering and Applications. Cambridge, MA, USA, November 2002.
- [OMG01] Object Management Group (OMG). *Unified Modeling Language Specification*, v1.4, 2001. At: <http://www.omg.org>
- [OMG03] Object Management Group (OMG). *Unified Modeling Language Specification*, v1.5, 2003. At: <http://www.omg.org>

- [OJ93] W. Opdyke and R. E. Johnson. *Creating abstract superclasses by refactoring*. In Proceedings of the 1993 ACM Conference on Computer Science(CSC'93), 66-73, Indianapolis, Indiana, USA, February 1993.
- [OPDY92] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.
- [PARN94] D. L. Parnas. *Software aging*. In Proceedings of 16th international conference on Software engineering (ICSE'94), 279-287, Sorento, Italy, May 1994.
- [PB90] S. L. Pfleeger and S. A. Bohner. *A framework for software maintenance metrics*. In Proceedings of IEEE Conference on Software Maintenance, (CSM90), 320-327, 1990.
- [PERR94] D. E. Perry. *Dimensions of software evolution*. In Proceedings of the International Conference on Software Maintenance (ICSM'94), 296-303, Victoria, BC, Canada, September 1994.
- [PERR98] D.E. Perry. *Generic descriptions for product line architectures*. In Proceedings of 2nd International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families, 51-56, Las Palmas de Gran Canaria, Spain, February 1998.
- [PG94] W. Pree, E. Gamma. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Reading, MA, 1994. ISBN: 0201422948
- [PIPK02] J. U. Pipka. *Refactoring in a "Test First"-World*. In Proceedings of 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP2002), Sardinia, Italy, May 2002. At: <http://www.agilealliance.org/articles/articles/JensUwePipka-RefactoringinaTestFirstWorld.pdf>
- [POST01] PostgreSQL Global Development Group. *PostgreSQL 7.2 Manuals*. 2001. Available at: <http://www.postgresql.org/docs/manuals/>
- [PR97] J. Philipps, B. Rumpe. *Refinement of information flow architectures*. In Proceedings of 1st IEEE International Conference on Formal Engineering Methods(ICFEM), 203-212, Hiroshima, Japan, November 1997.

- [PR01] I. Philippow, M. Riebisch. *Systematic definition of reusable architectures*. In Proceedings of 8th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'01), 128-135, Washington, DC, USA, April 2001.
- [PR04] I. Pashov, M. Riebisch. *Using feature modeling for program comprehension and software architecture recovery*. In Proceedings of 11th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'04), 406-418, Brno, Czech Republic, May 2004.
- [PREE92] R. Pressman. *Software Engineering A Practitioners Approach*. McGraw Hill. New York. 1992. ISBN:0070508143
- [PREE94] W. Pree. *Meta patterns – a means for capturing the essential of reusable object-oriented design*. In Proceedings of 8th European Conference on Object-Oriented Programming (ECOOP'94), 150-162, Bologna, Italy, July 1994.
- [PREE99] W. Pree. *Hot-Spot-Driven Development, Building Application Frameworks: Object-Oriented Foundations of Framework Design*. M. E. Fayad, D. C. Schmidt, R. E. Johnson (eds), 379-393, John Wiley & Sons, NY, 1999. ISBN: 0471248754
- [PRIE89] R. Prieto-Diaz. *Classification of reusable modules*. In Software Reusability: Concepts and Models, J. B. Ted and J. P. Alan,(eds), 99-123, Addison-Wesley, Reading, MA, 1989. ISBN: 0201080176
- [PROS99] J. Prosise. *Programming Windows with MFC*. 2nd Edition. Microsoft Press, 1999. ISBN: 1572316950
- [PTA94] C. Potts, K. Takahashi, and A. Anton. *Inquiry-based requirements analysis*. IEEE Software 11,2, 21-32, 1994.
- [PW92] D. E. Perry, A. L. Wolf. *Foundations for the Study of Software Architecture*. ACM SIGSOFT Software Engineering Notes 17,4,40-52, 1992.
- [PYK+97] J.M. Patel, J. Yu, N. Kabra, et al. *Building a scalable Geo-Spatial database system: technology, implementation and evaluation*. In Proceedings of 1997 SIGMOD Conference, 336-347, Tucson, Arizona, AZ, May 1997.

- [RAB96] B. Regnell, M. Andersson, and J. Bergstrand. *A hierarchical use case model with graphical representation*. In Proceedings of 2nd International Symposium on Engineering of Computer-Based Systems, 270-277, IEEE Computer Society Press, 1996.
- [RAMA96] R. Ramakrishnan. *The Minibase Home Page*. 1996. At: <http://www.cs.wisc.edu/coral/minibase/minibase.html>
- [RATO03] Rational Software Corporation. *The Object Constraint Language (OCL)*. 2003. At: <http://www.rational.com/uml/documentation.html>
- [RATU03] Rational Software Corporation. *Unified Modeling Language (UML)*, v1.5, 2003. At: <http://www.rational.com/uml/documentation.html>
- [RB98] A. Russo, B. Nuseibeh, J. Kramer. *Restructuring requirements specifications for managing inconsistency and change: A case study*. In Proceedings of 3rd International Conference on Requirements Engineering (ICRE'98), 51-61, Colorado Springs, CO, USA, April 1998.
- [RB00] V.T. Rajlich, K.H. Bennett. *A staged model for the software life cycle*. IEEE Computer, 33, 7, 66-71, July 2000.
- [RB03] K. Rui and G. Butler. *Refactoring use case models: the metamodel*. In Proceedings of 25th Australasian Computer Science Conference (ACSC2003), 301-308, Adelaide, Australia, February 2003.
- [RBCM91] D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro. *Approaches to program comprehension*. The Journal of Systems and Software 14,2,79-84, February 1991.
- [RBGM00] D. Riehle, R. Brudermann, T. Gross, K. Mätzel. *Pattern density and role modeling of an object transport service*. ACM Computing Surveys, 32, 1, No. 10, March 2000.
- [RBJ97] D. Roberts, J. Brant, and R. E. Johnson. *A refactoring tool for smalltalk*. Theory and Practice of Object Systems 3,4,253-263, 1997.
- [RBSF00] M. Riebisch, K. Böllert, D. Streitferdt, B. Franczyk. *Extending the UML to Model System Families*. 5th International Conference on Integrated

- Design and Process Technology (IDPT), Dallas, Texas, USA, June 2000. At: <http://www.theoinf.tu-ilmenau.de/~streitdf/TheHome/own/data/idpt2000-paper.pdf>
- [RBSP02] M. Riebisch, K. Böllert, D. Streitferdt, I. Philippow. *Extending feature diagrams with UML multiplicities*. 6th International Conference on Integrated Design and Process Technology (IDPT), Pasadena, California, USA. June 2002. At: <http://www.theoinf.tu-ilmenau.de/riebisch/publ/IDPT2002-paper.pdf>
- [RE93] B. Ramesh and M. Edwards. *Issues in the development of a requirements traceability model*. Proceedings of IEEE International Symposium on Requirements Engineering, (RE'93), 256-259. San Diego, CA, USA, January 1993.
- [REGN99] B. Regnell. *Requirements Engineering with Use Cases: a Basis for Software Development*. PhD thesis, Lund University, 1999.
- [RG93] K. S. Rubin and A. Goldberg. *Getting to why*. Journal of Object-Oriented Programming 6, 4, 513, 1993.
- [RG98] D. Riehle and T. Gross. *Role model based framework design and integration*. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, 117-133. Vancouver, Canada, October 1998.
- [RG00] R. Ramakrishnan, J. Gehrke. *Database Management Systems 2nd edition*. McGraw Hill, UK, 2000. ISBN: 0072465352
- [RJ97] D. Roberts and R. Johnson. *Patterns for Evolving Frameworks*, Pattern Languages of Program Design 3, R. C. Martin, D. Riehle, and F. Buschmann (eds), 471-486. Addison-Wesley, Reading, MA, 1997. ISBN: 0201310112
- [RJ01] B. Ramesh, M. Jarke. *Toward reference models for requirements traceability*. IEEE Transactions on Software Engineering, 27, 1, 58-93, January 2001.
- [RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA, 1999. ISBN: 020130998X
- [ROBE99] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

- [ROYC70] W. W. Royce. *Managing the development of large software systems: concepts and techniques*. In Proceedings of IEEE Western Electronic Show and Convention (WESCON'70), 1-9, Los Angeles, CA, USA, August 1970.
- [RRB03] K. Rui, S. Ren, G. Butler. *Refactoring use case models: a case study*. In Proceedings of 5th International Conference on Enterprise Information Systems (ICEIS 2003), 239-244, Angers, France, April 2003.
- [RSSS94] R. Ramakrishnan, D. Srivastava, S. Sudarshan, P. Seshadri. *The CORAL deductive system*. The VLDB Journal 3,2,161-210, 1994.
- [RUSS91] V. F. Russo. *An Object-Oriented Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, 1990.
- [SAF03] S. A. Sherba, K. M. Anderson, M. Faisal. *A framework for mapping traceability relationships*. In Proceedings of 3rd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'03), 32-39, Montreal, Quebec, Canada, October 2003.
- [SB99] M. Svahnberg, J. Bosch. *Evolution in software product lines: two cases*. Journal of Software Maintenance, 11, 6, 391-422, 1999.
- [SB00] M. Svahnberg and J. Bosch. *Issues concerning variability in software product lines*. In Proceedings of 3rd International Workshop on Software Architectures for Product Families, (IW-SAPF-3), 146-157, Berlin, German, March 2000.
- [SCHM97] H. A. Schmid. *Systematic framework design by generalization*. Communications of the ACM, 40, 10, 48-51, 1997.
- [SCHE98] A. W. Scheer. *Business Process Engineering: Reference Models for Industrial Enterprises*. Study Edition, Springer-Verlag, NY, 1998. ISBN: 3540638679
- [SCHM00] K. Schmid. *Scoping software product lines*, Software Product Lines, Experience and Research Directions. P. Donohoe (ed), 513-532, Kluwer Academic Publisher, AH, Netherlands, 2000. ISBN: 0792379403
- [SDKD03] M. J. Smith, R. G. Dewar, K. Kowalczykiewicz, D. Weiss. *Towards automated change propagation: the value of traceability*. Technical

- Report HW-MACS-TR-0002, Heriot-Watt University, 2003. Available at:
<http://www.macs.hw.ac.uk:8080/techreps/docs/files/HW-MACS-TR-0002.pdf>
- [SEI94] Software Engineering Institute. *Software Process Maturity Questionnaire, Capability Maturity Model*, v1.1.0. Pittsburgh. 1994.
- [SEI02] Software Engineering Institute. *Capability Maturity Model Integration (CMMI)*, V1.1. Pittsburgh. 2002.
- [SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on An Emerging Discipline*. Prentice Hall, NJ, 1996. ISBN: 0131829572
- [SIMO97] M. Simos. *Organization domain modeling and OO analysis and design: distinctions, integration, new directions*. In Proceedings of 3rd Conference on SmallTalk and Java in Industry and Education (STJA'97), 126132, Erfurt, Germany, September 1997.
- [SLMD96] P. Steyaert, C. Lucas, K. Mens, T. D'Hondt. *Reuse contracts: managing the evolution of reusable assets*. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA96), 268-285, San Jose, California, USA, October 1996.
- [STAR96] STARS. *Organization Domain Modeling (ODM) Guidebook*, v2.0. Technical report, Lockheed Martin Tactical Defense Systems, 1996.
- [STOR02] S. Storkel. *An introduction to the Eclipse IDE*. At:
<http://www.onjava.com/pub/a/onjava/2002/12/11/eclipse.html>
- [STRO97] B. Stroustrup. *The C++ Programming Language*. 3rd Edition. Addison-Wesley, Reading, MA, 1997. ISBN: 201889544
- [SUMM00] I. Sommerville. *Software Engineering*. 6th Edition, Addison-Wesley, Reading, MA, 2000. ISBN: 020139815X
- [SWAN76] E. B. Swanson. *The dimensions of software maintenance*. In Proceedings of 2nd IEEE International Conference on Software Engineering, 492-497, San Francisco, California, USA, October 1976.

- [TALI95] The Taligent Reference Library. *The Power of Frameworks: For Windows and OS/2 Developers*. Addison-Wesley, Reading, MA, 1995. ISBN: 0201483483
- [TB95] L. Tokuda and D. Batory. *Automating software evolution via design pattern transformations*. In Proceedings of 3rd International Symposium on Applied Corporate Computing, Monterrey, Mexico, October 1995.
- [TB99] L. Tokuda and D. Batory. *Automating three modes of evolution for object-oriented software architectures*. In Proceedings of 5th USENIX Conference on Object-Oriented Technologies (COOTS'99), 189-202, San Diego, California, USA, May 1999.
- [TB01] L. Tokuda, D. Batory. *Evolving object-oriented designs with refactorings*. Journal of Automated Software Engineering, 8,89–120, 2001.
- [TILB89] A. J. Tilbury. *Enabling software traceability*. IEEE Colloquium on the Application of Computer Aided Software Engineering Tools, 7/17/4, London, UK, 1989.
- [THOM04] D. Thomas. *MDA: Revenge of the Modelers or UML Utopia?* IEEE Software 21,3,15-17, 2004.
- [TM87] W. M. Turski, T. S. E. Maibaum. *The Specification of Computer Programs*. Addison-Wesley. Reading, MA, 1987. ISBN: 0201142260
- [TM03] T. Tourwé and T. Mens. *Identifying refactoring opportunities using logic meta programming*. In Proceedings of 7th European Conference on Software Maintenance and Reengineering, 91-100, Benevento, Italy, May 2003.
- [TOK99] L. Tokuda. *Evolving object-oriented architectures with refactorings*. In Proceedings of ASE-99, The 14th Conference on Automated Software Engineering. 174-182, Cocoa Beach, Florida, USA, October 1999.
- [TOKU99] L. Tokuda. *Design Evolution with Refactorings*. PhD thesis, University of Texas at Austin, 1999.
- [TRAC88] W. Tracz. *Software Reuse: Emerging Technology*. IEEE Computer Society Press, CA, USA, 1988. ISBN:0818608463

- [TRAC94] W. Tracz. *Domain-specific software architecture (DSSA) frequently asked questions (FAQ)*. SIGSOFT Software Engineering Notes 19,2,5256, April 1994.
- [TRAC95] W. Tracz. *DSSA (domain-specific software architecture): pedagogical example*. SIGSOFT Software Engineering Notes 20,3,49-62, July 1995.
- [TRUS99] L. Trussell. *Essential software development methodology*. IEEE Winter Meeting, January 1999.
- [TURS00] W. M. Turski. *Essay on software engineering at the turn of century*. Fundamental Approaches to Software Engineering, 3rd International Conference, T. S. E. Maibaum (ed). 1-20, Springer-Verlag, Berlin, German, March 2000.
- [VAM+98] A. D. Vici, N. Argentieri, A. Mansour, M. d'Alessandro, J. Favaro. *FO-DACom: an experience with domain analysis in the Italian telecom industry*. In Proceedings of 5th International Conference on Software Reuse, 166-175, Victoria, BC, Canada, June 1998.
- [VSF97] A. Valerio, G. Succi, M. Fenaroli. *Domain analysis and framework-based software development*. ACM Special Issue on Frameworks and Patterns in Software Reuse, 5, 2, 4-15, September 1997.
- [WEI04] Y. Wei. *Refactoring Use Case Models on Episodes*. Master Thesis, Concordia University, 2004.
- [WGM88] A. Weinand, E. Gamma, and R. Marty. *ET++ - an object-oriented application framework in C++*. In Proceedings of Object-Oriented Programming Systems, Languages, and Applications Conference (OOPLSA'88), 46-57, San Diego, California, September 1988.
- [WH02] C. V. Westhuizen, A. V. Hoek. *Understanding and propagating architectural changes*. In Proceedings of IFIP 17th World Computer Congress - TC2 Stream/3rd IEEE/IFIP Conference on Software Architecture (WICAS3), 95-109, Deventer, The Netherlands, August 2002.
- [WL99] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Reading, MA, 1999. ISBN: 0201694387

- [WW93] D. A. Wilson, S.D. Wilson. *Writing frameworks – capturing your expertise about a problem domain*. In tutorial notes, 8th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPLSA'93), Washington, DC, USA, September 1993.
- [ZC03] D. Zubrow, G. Chastek. *Measures for software product lines*. Software Engineering Measurement and Analysis Initiative, Software Engineering Institute, CMU/SEI-2003-TN-031, October 2003.
- [ZSPK03] A. Zisman, G. Spanoudakis, E. Pérez-Miñana, P. Krause. *Tracing software requirements artefacts*. In Proceedings of 2003 International Conference on Software Engineering Research and Practice (SERP'03), 448-455, Las Vegas, Nevada, USA, June 2003.