UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

Dissertation

submitted to the

Faculty of Computer Science and Mathematics
University of Passau

Enhancing Information Systems with Event-Handling - A Non-Invasive Approach

Michael Guppenberger

May 5, 2010

Supervisor: Prof. Dr. Burkhard Freitag

Dissertation for the acquisition of the degree of a doctor in natural sciences at the Faculty of Mathematics and Computer Science, University of Passau

1st reviewer: Prof. Dr. Burkhard Freitag, University of Passau
2nd reviewer: Prof. Dr. Harald Kosch, University of Passau

Abstract

Due to the immense advance of widely accessible information systems in industrial applications, science, education and every day use, it becomes more and more difficult for users of those information systems to keep track with new and updated information. An approach to cope with this problem is to go beyond traditional search facilities and instead use the users' profiles to monitor data changes and to actively inform them about these updates - an aspect that has to be explicitly developed and integrated into a variety of information systems. This is traditionally done in an individual way, depending on the application and its platform.

In this dissertation, we present a novel approach to model the semantic interrelations that specify *which users* to inform about *which updates*, based on the underlying model of the respective information system. For the first time, a meta-model that allows information system designers to tag an arbitrary data model and thus specify the event-handling semantics is presented. A formal specification of how to interpret meta-models to determine the receivers of the events completes the presented concept.

For the practical realization of this new concept, model driven architecture (MDA) shows to be an ideal technical means. Using our newly developed UML profile based on data-modelling standards, an implementation of the event-handling specification can automatically be generated for a variety of different target platforms, like e.g. relational databases, using triggers. This meta-approach makes the proposed solution ideal with respect to maintainability and genericity. Our solution significantly reduces the overall development efforts for an event-handling facility. In addition, the enhanced model of the information system can be used to generate an implementation that also fulfils non-functional requirements like high performance and extensibility.

The overall framework, consisting of the domain specific language (i.e. the meta-model), formal and technical transformations of how to interpret the enhanced information system model and a cost-based optimizing strategy, constitutes an integrated approach, offering several advantages over traditional implementation techniques: our framework can be applied to new information systems as well as to legacy applications

without having to modify existing systems; it offers an extensible, easy-to-use, generic and thus re-usable solution and it can be tailored to and optimized for many use cases, as the practical evaluation presented in this dissertation verifies.

Zusammenfassung

Bedingt durch die immer stärkere Durchdringung rechnergestützter Informationssysteme in Industrie, Forschung, Ausbildung und anderen Bereichen des täglichen Lebens wird es für Anwender immer schwieriger, für sie relevante Änderungen an den dort gespeicherten Datenbeständen nachzuverfolgen. Dem wird häufig dadurch begegnet, dass über die Fähigkeiten traditioneller Suchmöglichkeiten hinaus gegangen wird und Profile der Anwender verwendet werden, um sie aktiv über relevante Änderungen zu informieren. Dieser Aspekt muss für unterschiedlichste Informationssysteme explizit entwickelt und integriert werden, zudem meist abhängig von der fachlichen Domäne der Anwendung und deren Plattform.

In dieser Dissertation präsentieren wir einen neuartigen Ansatz, mit dessen Hilfe die semantischen Vorgaben, *welche Anwender* über *welche Änderungen* informiert werden sollen, ausgehend vom zugrunde liegenden Datenmodell der Anwendung des jeweiligen Systems modelliert werden können. Erstmalig wird ein Meta-Modell vorgestellt, das Entwicklern und Architekten ermöglicht, ein beliebiges Modell eines Informationssystems mit zusätzlichen Informationen auszuzeichnen und damit die Semantik der Event-Handling-Komponente vorzugeben. Zudem wird ein formales Konzept präsentiert, das spezifiziert wie diese Auszeichnungen für die Bestimmung der Informationsempfänger zu interpretieren sind.

Im Hinblick auf die Realisierung dieses Konzepts erweist sich Model Driven Architecture (MDA) als ideales technisches Mittel. Mit Hilfe eines eigens entwickelten UML Profils, das sich auf existierende Standards zur Datenmodellierung stützt, kann automatisch eine Implementierung der Event-Handling-Komponenten für eine Vielzahl unterschiedlichster Zielplattformen generiert werden. Als Beispiel wäre die Verwendung relationaler Datenbanken zusammen mit Datenbanktriggern zu nennen. Dieser Ansatz stellt eine ideale Lösung im Hinblick auf Wartbarkeit und Allgemeingültigkeit dar, wodurch auch der Entwicklungsaufwand minimiert wird. Zudem bietet unser Ansatz auch die Möglichkeit, bei der Implementierung dieser Komponente auch nicht-

funktionale Anforderungen - wie beispielsweise möglichst optimale Performanz und Erweiterbarkeit - zu erfüllen.

Das hier präsentierte Framework, bestehend aus der domänen-spezifischen Sprache (in Form des Meta-Modells), den formalen und technischen Transformationsvorschriften für die Interpretation der Spezifikation sowie einer kostenbasierten Optimierungsstrategie, stellt einen integrierten Ansatz dar, der im Vergleich zu traditionellen Ansätzen einige Vorteile bietet: so kann dieser Ansatz ohne Modifikation existierender Systeme verwendet werden, stellt eine erweiterbare, einfach benutzbare, und zugleich wiederverwendbare Lösung dar und kann für beliebige Anwendungsfälle maßgeschneidert und optimiert werden, wie die Evaluation unserer Lösung anhand echter Szenarien in dieser Dissertation zeigt.

To my grandmother.

Acknowledgements

First and foremost, I would like to thank Prof. Dr. Burkhard Freitag for giving me the opportunity to write this dissertation during my employment at his institute and for supervising me in the course of my research. I am grateful for his support during the many years, applying adequate pressure when needed, whilst accepting delays when other aspects of my life took my whole attention. His guidance and advice were always very helpful and pointed me in the right direction. I would also like to thank Prof. Dr. Harald Kosch for acting as a second reviewer and for his positive feedback.

Furthermore, this work would not have been finished without the help of many of my former and current colleagues: I would especially like to thank Ursula Falenczyk, Dr. Uli Zukowski, Thomas Hackl and Guido Lenk for their manifold assistance and advice. My gratitude also goes to Dr. Armin Bender and his team for their support with the print-production of this thesis, to Max Reiter for taking over some of my duties at *msg systems* during the final phase of my work on the dissertation and to Karin Hemetsberger for carefully proofreading the pre-final version of this thesis.

Writing this dissertation also would not have been possible without appropriate support in my private life. It is impossible to name all of my friends who somehow supported me during the work on my thesis, but I would especially like to thank Andrea Schnabl, who accompanied me throughout the initial phase of my work and encouraged me to proceed when I was about to quit, as well as Andreas Löhr who often gave valuable feedback and answered many of my questions.

Finally, it is a pleasure to express my gratitude to all my family members who helped me in getting this far. Sincere thanks go out to my grandparents, my grand aunt and my late grand uncle for their support during my studies, to my sister for cheering me up when I had bad times, and - most important - to my parents, who supported me wherever they could, not only regarding my professional career. I especially want to thank Heike, Celina-Marie and Lucie for their understanding, for taking care of everyday affairs and for their continuous support and care over the years.

Contents

I Problem Statement

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Contribution of our Work . . . 7
1.3 Overview . . . 7

2 Problem Statement and Requirements Analysis **9**
2.1 Use Cases . . . 9
2.1.1 Stud.IP . . . 10
2.1.1.1 System Overview . . . 10
2.1.1.2 A Spotlight on *Stud.IP*'s Notification System . . . 11
2.1.1.3 Software-Technological Analysis . . . 11
2.1.1.4 Essential Cognitions . . . 19
2.1.2 InfoWiss . . . 23
2.1.2.1 System Overview . . . 23
2.1.2.2 A Spotlight on *InfoWiss*' Event-Handling . . . 23
2.1.2.3 Software-Technological Analysis . . . 24
2.1.2.4 Essential Cognitions . . . 28
2.1.3 Further Use Cases in Brief . . . 30
2.2 Inferred Requirements . . . 31
2.2.1 Semantic Requirements . . . 31
2.2.2 Technical Requirements . . . 33
2.2.3 Software-Engineering Requirements . . . 34
2.3 Summary . . . 35

3 Existing Approaches **37**
3.1 Classification Scheme for Publish/Subscribe Systems . . . 37

3.2 Overview of Existing Approaches 43
3.2.1 Research Projects 43
3.2.1.1 Applications 43
3.2.1.2 Meta-Modelling and Model Transformation 44
3.2.1.3 Subscription Specification 45
3.2.1.4 Event Matching 46
3.2.1.5 Matching Optimization 47
3.2.1.6 Event Detection 48
3.2.1.7 Data Storage 49
3.2.1.8 Subscription and Notification Storage 49
3.2.1.9 Publication Interface 49
3.2.1.10 Notification Delivery 49
3.2.1.11 Security 50
3.2.2 Commercial Systems 50
3.2.3 Comparison Against Requirements 51
3.3 Summary 53

II The Non-Invasive Approach

4 Solution Overview and Generic Architecture 57
4.1 Motivation for the Generative Approach 57
4.2 The Declarative-Generative Approach 60
4.2.1 Central Data Storage 61
4.2.2 Information System's Data Model as a Basis 61
4.2.3 Generic Meta-Model of Arbitrary Data Models 61
4.2.4 Enhancement of the Meta-Model by a Generic Event-Handling Meta-Model 61
4.2.5 Enrichment of the Data Model 63
4.2.6 Transformation of the Enriched Data Model into Event-Handling Code 63
4.3 Genericity and Dimensions of Abstraction 64
4.4 Details on the Architecture's Components 65
4.4.1 Models 65
4.4.2 Event-Handling Data Access Layer 66
4.4.3 Event Detection 67
4.4.4 Scenario Monitoring 67
4.4.5 Publish/Subscribe Component 67
4.4.6 Optimizer/Generator 68

4.4.7 Legacy Application 68
4.5 Dynamic View on the System's Lifecycle 68
4.5.1 Designtime Lifecycle 68
4.5.2 Runtime Lifecycle 70
4.6 Summary 71

5 Conceptualization of Data and Event Models 73
5.1 Representation of Data Model and System Instance 74
5.1.1 Data Model 74
5.1.2 System Instance 75
5.2 Event-Handling Constructs and Formal Event Model 77
5.2.1 Subscribers 77
5.2.2 Subscribables 78
5.2.3 Observed Attributes 79
5.2.4 Implicit Subscriptions 79
5.2.5 Event-Propagating Associations 80
5.2.6 Explicit Subscriptions 81
5.3 Overlays 81
5.4 Graph Representations for Data Model, Overlay and Instance 83
5.4.1 Graph Representation of Data Model 83
5.4.2 Graph Representation of System Instance 83
5.4.3 Graph Representation of Overlays 84
5.4.4 Path Descriptions 85
5.4.5 Path Instances 85
5.5 Interpretation 87
5.5.1 Definitions 87
5.5.2 Subscribers to Inform about Updates 89
5.5.2.1 Implicit Subscribers 90
5.5.2.2 Explicit Subscribers 91
5.6 Real-Life Example 92
5.6.1 Sample Scenario 92
5.6.2 Formal Representation 94
5.6.3 Interpretation of the Event-Handling Specification 96
5.7 Discussion on Design Decisions 100
5.7.1 Handling Inheritance 100
5.7.2 Non-Transitive Implicit Subscriptions 100
5.7.3 Benefit of Overlays 101
5.8 Summary 102

6 Generic Trigger Generation **103**
6.1 Overview of the Generation Algorithm 103
6.2 Generation Algorithm in Detail 104
6.3 Sample Generation Process 110
6.4 Qualitative Analysis 112
6.4.1 Correctness 112
6.4.2 Avoidance of Event Cascades 114
6.4.3 Discussion on Cycles 115
6.5 Summary 116

7 Generic Optimization Strategy **117**
7.1 Optimization Idea 117
7.1.1 Computation of Matching Paths 118
7.1.2 Optimization by Precomputing Matching Paths 118
7.2 Strategy for the Usage of Event Propagation Indices 119
7.3 Cost Model 121
7.3.1 Costs of Event Propagation Indices 121
7.3.2 Costs of Paths 122
7.4 Probabilities 123
7.5 Expected Costs of Path Descriptions 125
7.6 Probability Models 126
7.6.1 Heuristic Probability Model 126
7.6.2 Designtime Probability Model 126
7.6.3 Empirical Probability Model 127
7.6.4 Comparison of Probability Models 127
7.7 Cost Models 128
7.7.1 Heuristic Cost Model 128
7.7.2 DBCostModel 132
7.7.3 Empirical Cost Model 132
7.8 Sample Comparison of Indexing Alternatives 133
7.8.1 Scenario 1 134
7.8.2 Scenario 2 135
7.8.3 Scenario 3 136
7.8.4 Scenario 4 137
7.8.5 Empirical Validation of the Results 138
7.8.6 Sophisticated Index Maintenance Algorithms 139
7.8.7 Results 140
7.9 Determining the Optimal Index Usage 141

7.10 Estimated Behaviour Depending on Path Length 142
7.10.1 Highly Connected Object Graphs 143
7.10.2 Sparsely Connected Object Graphs 145
7.10.3 Consequences 146
7.11 Summary 147

III The Model Driven Implementation for Active Databases

8 Technology Selection 151
8.1 Motivation for Technology Selection 151
8.2 Model Driven Architecture 152
8.2.1 MDA in General 152
8.2.1.1 The MDA Development Life Cycle 152
8.2.1.2 Automation of the Transformation Steps 154
8.2.1.3 Building Blocks of MDA 155
8.2.2 AndroMDA 159
8.3 Active Database Technology 160
8.4 Materialized Views 162
8.5 Summary 164

9 Reference Architecture and Implementation 165
9.1 Substantiating the Abstraction Layers 165
9.1.1 UML Profiles and MDA 166
9.1.2 Transforming Models into Triggers 167
9.1.2.1 Using Materialized Views for Optimization Purposes . 167
9.1.2.2 Cost Model for Relational Databases 168
9.2 System Architecture Components in Detail 168
9.2.1 Event-Handling Data Access Layer 168
9.2.2 Event Processing using Triggers 170
9.2.2.1 Database Tables to Monitor 170
9.2.2.2 Determining Relevant Subscribers using Triggers . . . 175
9.2.2.3 Example Without Optimization 177
9.2.2.4 Processing Notification Entries 183
9.2.2.5 Optimization 183
9.2.2.6 Inheritance Revisited 191
9.3 Summary 193

10 From UML Models to Optimized Triggers **195**
10.1 UML Profile . . . 195
10.1.1 Object-Relational Mapping . . . 196
10.1.2 Event-Handling Profile Elements . . . 198
10.2 Model Template . . . 200
10.3 Metafacades and Transformation Helper Classes . . . 201
10.3.1 AndroMDA Base Metafacades . . . 201
10.3.2 Event-Handling Metafacades . . . 204
10.3.3 Transformation Helper Classes . . . 206
10.4 MDA Transformations . . . 208
10.5 Correctness . . . 211
10.6 Comprehensive Sample Model . . . 212
10.7 Summary . . . 224

IV Résumé

11 Critical Evaluation **229**
11.1 Evaluation of the Prototypic Implementation . . . 229
11.1.1 Software Quality According to ISO 9126 . . . 230
11.1.1.1 Functionality . . . 230
11.1.1.2 Reliability . . . 231
11.1.1.3 Usability . . . 231
11.1.1.4 Efficiency . . . 232
11.1.1.5 Maintainability and Portability . . . 232
11.1.2 A Detailed View on Performance and Scalability . . . 233
11.1.2.1 Performance and Scalability in General . . . 233
11.1.2.2 Benefit of Optimization . . . 241
11.1.2.3 Real-Life Scenario . . . 245
11.1.2.4 Overall Performance Results . . . 249
11.1.3 Applicability in Distributed Environments . . . 249
11.1.4 Technological Alternatives . . . 251
11.1.4.1 Alternatives to MDA . . . 251
11.1.4.2 Alternatives to Database Triggers . . . 252
11.1.4.3 Alternatives to Materialized Views . . . 253
11.2 Rating of the Event-Handling Concept . . . 253
11.2.1 Fulfilment of Basic Postulated Requirements . . . 253
11.2.1.1 Functional Adequacy . . . 254
11.2.1.2 Technical / Architectural Adequacy . . . 257

11.2.1.3 Software-Technological Adequacy 258
11.2.2 Technological and Conceptual Alternatives 259
11.3 Review of the Generative Approach in General 260
11.3.1 Assets and Drawbacks 260
11.3.2 Factors for the Successful Usage of the Generative Paradigm . 261
11.4 Summary 262

12 Contribution and Open Ends 265
12.1 Contribution of Our Work 265
12.2 Open Ends 267
12.2.1 Technological Improvements 267
12.2.2 Conceptual Improvements 269
12.2.3 Visionary Ideas 270
12.3 Summary 271

A Bibliography 273

B Technical Details 287
B.1 Automatically Refreshing Materialized Views in DB2 287
B.2 Maintaining Event Propagation Indices with SQL Server 288

C List of Figures 291

D Listings 295

Part I

Problem Statement

"The beginning is the most important part of the work."

Plato

1 Introduction

1.1 Motivation

During the last 20 years, computer-based information systems have pervaded our everyday lives more and more: almost any information that can be stored and maintained is managed using electronic storage facilities. The applications range from small, integrated information systems like personal digital assistants (PDAs) up to world-wide large-scale databases, for instance vital data pools of global enterprises. Whilst personal information systems are designed to be used by a single user only, global information systems are characterized by the concurrent usage by many different stakeholders. The usage can be differentiated into two different use case categories: clients *modifying* or *creating* data vs. clients *consuming* data, as visualized in figure 1.1.

Both types of users, depicted by ClientA and ClientB, use the same information system. However, due to different responsibilities, each of them works on different subsets of data. For instance, looking at a university information system, members of the various chairs edit information about lectures, whilst administrative employees work with the same system, modifying time schedules and room-planning information. In spite of the separate responsibilities and tasks, the parts of the data the different users work with,

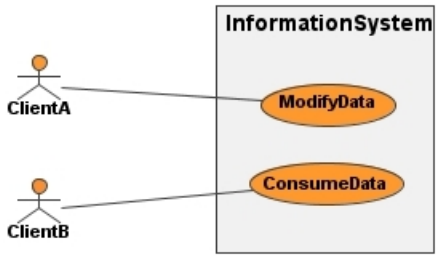


Figure 1.1: Use case: Clients accessing and modifying data

which we call data clouds, overlap (cf. figure 1.2). Recalling the university system, room planners might be interested in updates of the number of attendants of various lectures, although the administration of lecture data is not their primary task.

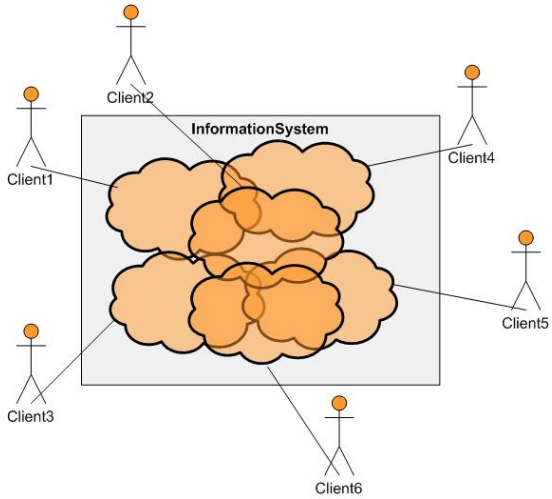


Figure 1.2: Overlapping of individually maintained data

This leads to an interesting problem: how can users stay up-to-date about modifications of data belonging to their responsibility, especially if these updates are performed spatially and temporally separate from their own location and time? It is obviously impossible to solve this task by regularly scanning the data pool and using traditional search functionality, trying to find relevant updates. Thus, event-handling solutions, actively and autonomously informing users, are required.

The demand for a solution to this problem gains even more importance with the further development of the World Wide Web: whilst web information traditionally used to be maintained by a small amount of editors and webmasters, it is the collaborative

approach of applications like Wikipedia [Wikb], Flickr [Yah], blogs, forums and many more (briefly named as the "Web 2.0" [O'R05]) that makes the WWW a collaborative work space for millions of users with millions of overlapping data clouds. Without tools helping the users to keep an overview of relevant updates, they would literally be lost in information space: either because of *not* recognizing updates that are important to them, or because of *being overwhelmed* by irrelevant update information. Challenges like this have already been foreseen more than ten years ago [BBC+98], where the inversion of the search paradigm has been demanded, leading to systems that actively notify their users: "Computers can augment human intelligence by [...] informing people when interesting things happen.".

Figure 1.3 visualizes the common requirements for such an event-handling system:

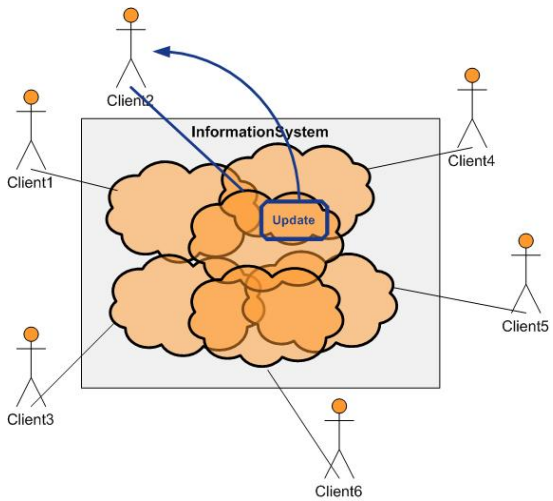


Figure 1.3: Requirements of an event-handling framework

- Specification of data clouds
  For every user, a possibility to specify the individual data cloud representing his or her responsibilities and/or interests has to be provided. As we will present later, this can be done based on two principles: On the one hand side, there are explicit specifications, i.e. the user himself tells the information system about the particular fragment of data he wants to be informed about. On the other hand, information about a user, which is already stored within the system, can be used in conjunction with knowledge about the semantic interdependences of the underlying data model to derive the user's data cloud. This knowledge can be used as an implicit specification of the user's interest.

- Determination of update events
  Updates, performed by any arbitrary system client, have to be monitored to be processed.

- Matching update events
  After an update has been detected, it has to be matched with the individual data clouds to determine all relevant users who have to be informed.

- Delivery of update events
  Finally, the recorded update has to be delivered to those users.

- Support for developers
  When considering the development and maintenance of such a system, a third group of stakeholders comes into play: the developers and administrators of an information system. Obviously, an approach to develop an event-handling system should provide a framework that can be used for a variety of different use cases. This framework should be easy to use, declarative, re-usable and generic.

- Applicability to legacy systems
  One last requirement serves an important demand: many legacy information systems are already running in many different fields of applications and do *not* yet support event-handling functionality. A generic framework should thus not only support developers of a new information system, but also allow the extension of existing systems with the needed notification components, ideally without having to significantly modify the legacy systems.

Although many existing software systems already provide such notification functionalities, there is, to our best knowledge, no comprehensive, simple and technologically mature approach yet that fulfills all the above-mentioned requirements. Thus, this dissertation aims to develop an integrated framework offering the possibility to enhance arbitrary information systems with components to monitor update events, evaluate them semantically and determine the target group that has to be informed about those respective updates.

1.2 Contribution of our Work

In this dissertation, we present a novel integrated, non-invasive and model-driven framework to develop event-handling systems. Our work consists of the following building blocks:

- An analysis of real-life use cases and existing notification approaches, leading to a set of requirements that have to be fulfiled
- A formal concept for notification semantics, based on a representation of structured information systems
- A domain specific language, representing the above-mentioned formal concept, to model event-handling functionalities for arbitrary applications
- A transformation from a declarative event-handling specification, based on this domain specific language, into executable, imperative code
- An implementation of this transformation based on Model Driven Architecture, which can be used for various target information systems
- A prototypic implementation using relational databases and database triggers
- An optimization strategy, based on a generic cost model, that can be used to speed up the evaluation of updates by precomputing parts of the notification results of an occured event

All these contributions are provided both as conceptual results, as well as in the form of a generic, prototypic application development framework. As we will show in this dissertation, this framework is the first to incorporate event-handling systems and the declarative, model-driven approach. Further on, we will show that all functional and non-functional requirements that have been identified during the analysis of various use cases are satisfied by our solution and that our generative, non-invasive approach is adequate.

1.3 Overview

This dissertation is organized as follows: chapter 2 contains a detailed analysis of several use cases to determine the semantical and technical requirements that have to be satisfied by an event-handling system. In chapter 3, we discuss existing approaches

in the field of event-handling systems and show that no appropriate solution exists, thus motivating us to develop the framework that is presented in this dissertation.

Part II deals with our non-invasive approach on a conceptual level: first, chapter 4 gives a bird's-eye overview of the solution, which is detailed in chapter 5, presenting the formal concept. The generic event-handler generation is then explained in chapter 6. This part concludes with a presentation of the optimization techniques that can be applied (chapter 7).

The model driven realization of the concept using active database technology, the second major contribution of this dissertation, is subject to part III. After a brief introduction to the underlying technologies (chapter 8), the implementation specific architecture, i.e. an instantiation of the generic system architecture, is presented in chapter 9. A realization of the generic transformation and the respective optimization techniques based on active database technology is then subject to chapter 10, which concludes the implementation-specific part of this dissertation.

Finally, part IV is dedicated to the evaluation and assessment of our non-invasive model driven approach (chapter 11) and a summary of the results of this dissertation, together with a list of open ends that could be dealt with in the course of any follow-up research efforts (chapter 12).

"An undefined problem has an infinite number of solutions."

Robert A. Humphrey

2

Problem Statement and Requirements Analysis

While the previous chapter motivated our work, the next pages will analyze the problem more precisely: we will present several existing software systems (with and without event-handling capabilities) and the respective use cases that led to the development of the solution that is presented in this dissertation. We dissect those use cases and collect their commonalities, which will later be used to develop a generic framework. Based on the use cases, we also deduct the requirements concerning an event-handling framework.

2.1 Use Cases

In the following, we will examine several information systems from various areas and analyze how they reflect different event-handling use cases.

2.1.1 Stud.IP

The first analyzed system is a german information system for universities called *Stud.IP* [Dat]. This system was introduced at the University of Passau during the three year research and development project *InteLeC* [Uni] during the years 2005 to 2008.

2.1.1.1 System Overview

Stud.IP is an open-source, web-based learning information system. Its main purpose is to coordinate and support the performance of courses and lectures at universities and other educational institutions. Therefore, it contains various functionalities:

- Model of the organizational structure
  Stud.IP stores a hierarchical representation of the unversity's organizational structure, i.e. the different faculties and their various sub-units, like chairs and administrative departments. All employees can be assigned to one or more of these organizational units.

- Lecture and event database
  All events that take place (lectures, exercises, talks, ...) can be administered using *Stud.IP*. They can be assigned to their respective lecturers and to the organizational units these lecturers belong to. Additionally, *Stud.IP* manages the time schedules of events, cares for the allocation of resources like rooms and maintains lists of all participants of a particular event.

- Lecture and event administration
  For every individual event, lecturers are supported in its implementation. All kinds of learning material (scripts, exercises, slides, ...) can be uploaded and thus be offered to the event participants. For communication purposes, per-event forums and chat rooms are provided. In addition, the individual dates and places when and where the event takes place can be maintained; updates of these dates and other topical information can be spread using the integrated news system.

- Further functionalities
  Although the above-mentioned functionalities make up the core of *Stud.IP*, there are many further capabilities, like the individual creation of lecture schedules for each student, manyfold communication features (chat rooms, wikis, messaging

system, ...), evaluation mechanisms, an integrated literature management system, and many more.

Due to the importance of a timely and purposeful delivery of information to the individual users, users of *Stud.IP* have an integrated event-handling system at their disposal. This functionality, which we will describe in the following, is an important part of *Stud.IP*.

2.1.1.2 A Spotlight on Stud.IP's Notification System

Stud.IP contains a simple event-handling and notification system:

- By default, every user who is associated with an event (for instance by attending a lecture, by organizing an appointment or by leading a discussion group), has an individual portal page with an overview of his or her events. An exemplary "My Lectures"-page is shown in figure 2.1.
- In his or her personal settings, every user can specify (for every event) if he or she wants to be informed about updates. By using the checkbox grid (highlighted in figure 2.2), it can be stated whether new or updated documents, posts, dates, news etc. should lead to a notification or not. A scheduled task then automatically sends emails to notify the users about the respective updates.
- In addition, the "My Lectures" screen displays highlighted icons if there have been any updates which have not yet been confirmed by the user, as the highlighted section in figure 2.1 shows.

These simple functionalities represent the notification component of *Stud.IP* and enable the system's users to stay up-to-date with respect to the events they might be interested in. In the following, we will take a closer look at the technical realization of this component.

2.1.1.3 Software-Technological Analysis

To analyze the realization of the notification component in *Stud.IP*, we recall the four aspects from section 1.1:

- Specification of data clouds
  Stud.IP strongly pre-structures the specification of interest: for every event the

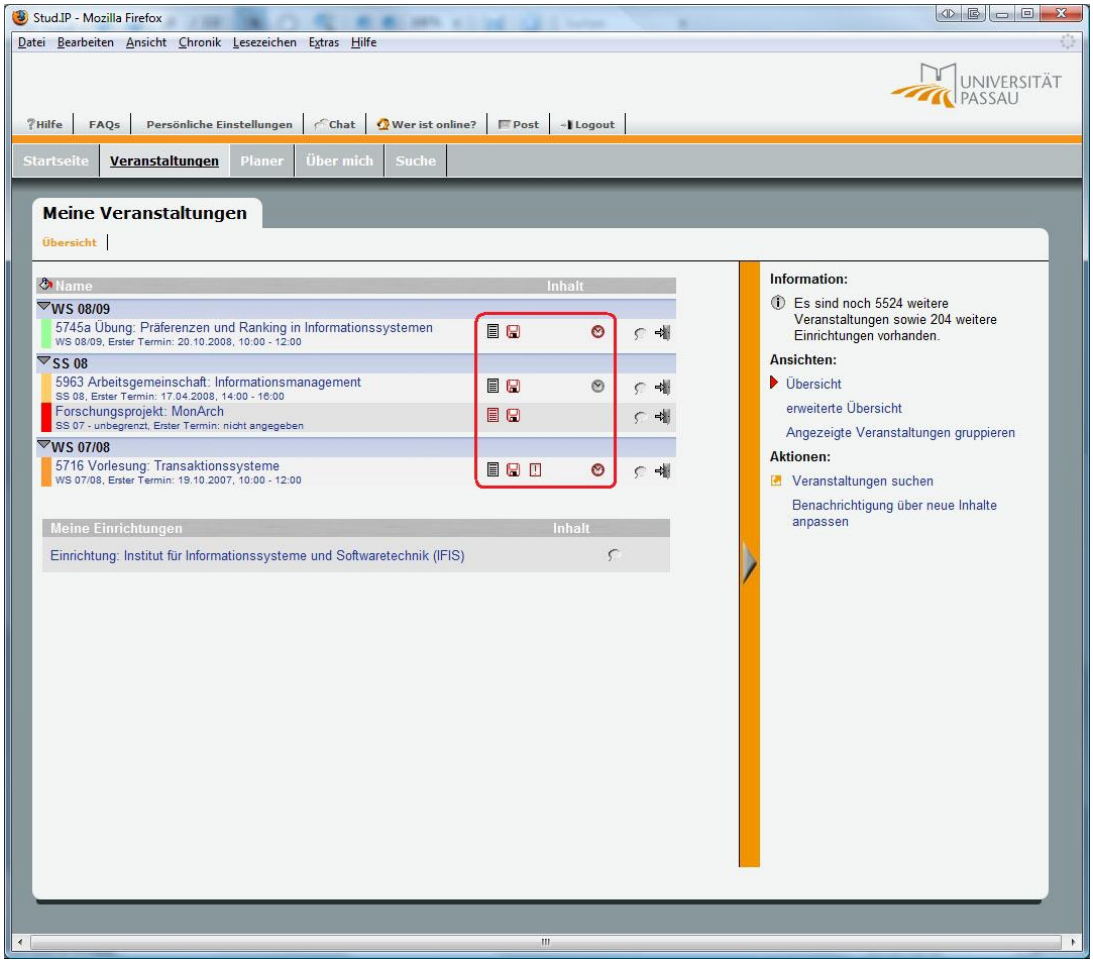


Figure 2.1: Screenshot: Lecture overview in *Stud.IP*

user takes part in, interest can be stated by checking or unchecking one of the predefined update types. Both the functionality to specifiy interest as well as the different update types are hard-coded; an individual interest matrix for every user stores the respective specification in the underlying database.

- Determination of update events
  The determination of update events, i.e. the computation of potential notifications, is also implemented unalterably in the source code: every time a user accesses his personal portal page, the modification dates of all relevant pieces of information are compared to the user's last access to these documents, forum

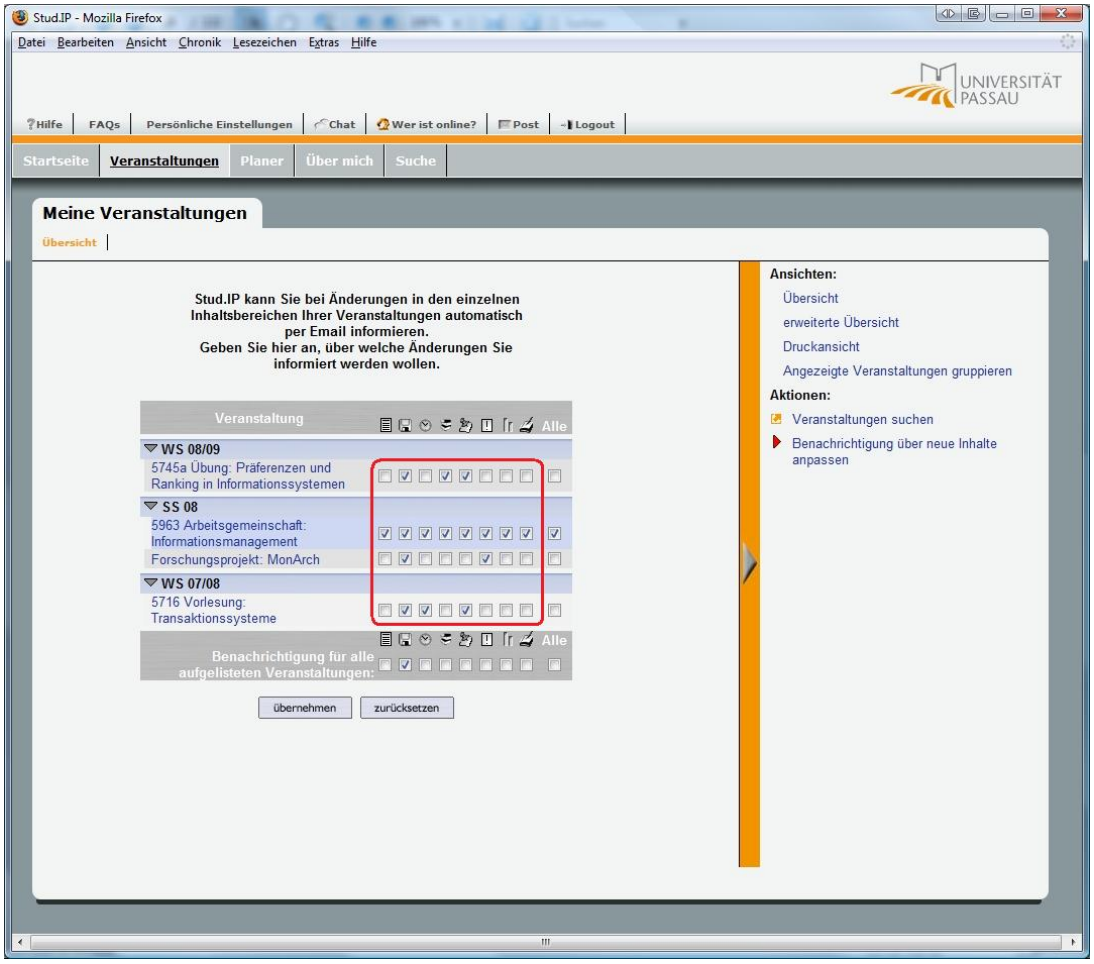


Figure 2.2: Screenshot: Notification settings in *Stud.IP*

posts, etc. If the modification date is newer than the last access, a highlighted icon is shown to inform the user about an update he has not yet noticed.

For the determination of email notifications, system administrators have to schedule a batch job that regularly compares modification dates against access dates. If a user has signaled interest to this kind of update by selecting the respective option in his personal settings, an email is automatically sent, containing short information about the updates.

- Matching update events
  Update events are matched against the user's profile (i.e. his or her data cloud)

by comparing the type and the associated event of the update to the user's interest matrix. If both match, the update is considered to be relevant and causes a notification. Again, this check is hard-coded and strongly coupled to the underlying data model of events and their documents.

- Delivery of update events
  Update events are delivered to users via two possible channels: on their portal page (by highlighting the respective icons) and by email. As we already stated, email is sent by a recurrent batch process (usually once a day).

To further analyze the implementation of the notification functionality, we take a closer look at the relevant sections of the data model.

Data Model Analysis Figure 2.3 shows an excerpt of *Stud.IP*'s data model,[1] focused on the tables and attributes that are relevant to the notification component.

The following tables have been examined in detail. Coloured tables represent entities that are mainly used for event-handling purposes, while the other tables contain central information in *Stud.IP*.

- Table auth_user_md5 contains information about the system's users, i.e. the receivers of potential update notifications.
- Table institut stores information about the different represented organizational units, i.e. the chairs, institutes, departments, etc.
- Users are assigned to organizational units via the association table user_inst.
- Table seminare is another central component of the data model, representing the different events taking place at the university: lectures, seminars, talks, and many more.
- Via table seminar_inst, events are assigned to particular institutions which are involved in the organization of the event.
- To manage documents that can be uploaded for every event, table dokumente is used, where every document can be assigned to a particular event using the attribute seminar_id.

[1] Since *Stud.IP* has initially been developed for an early version of MySQL that did not support foreign key constraints, the foreign keys and associations between entities have been added to this diagram as the result of a semantic reverse-engineering process.

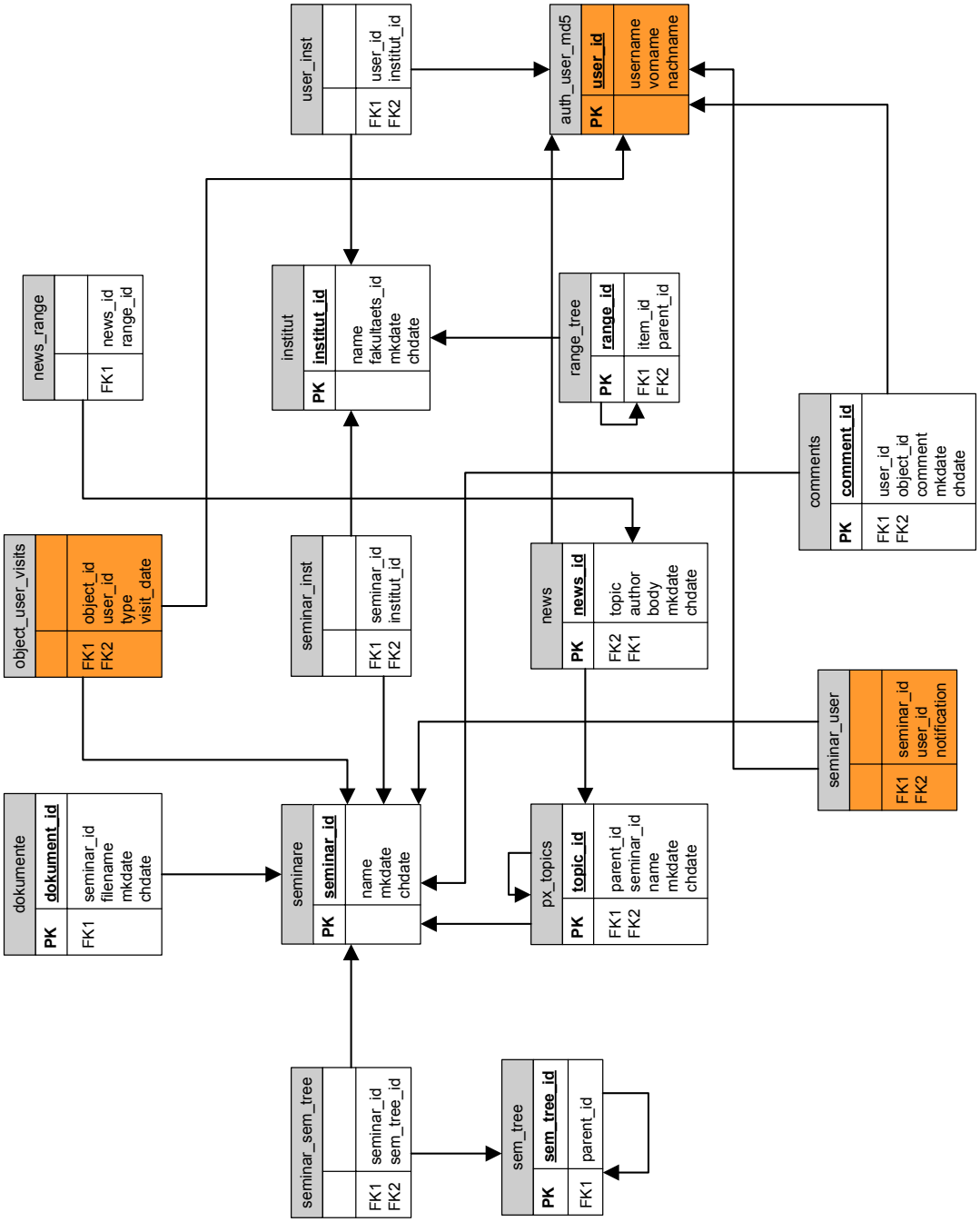


Figure 2.3: Excerpt from *Stud.IP*'s data model

- Since institutions represent the organizational hierarchy of the university, they are organized in the hierarchical structure range_tree, using the attribute parent_id as a pointer to the parent organizational unit. Via the attribute range_id, institutions are assigned to the structural unit.
- Similarly, events can be organized to represent hierarchical structures concerning their contents. Table sem_tree with attribute parent_id as a parental pointer represents this structure. Via table seminar_sem_tree, events are assigned to the respective structural level.
- Another central table is px_topics, storing a hierarchical system of topics, i.e. semantical units, connected to each other using the attribute parent_id. Each topic can be assigned to an event, for instance to specify that a lecture deals with the particular topic.
- news represents another central functionality of *Stud.IP*: users (i.e. authors) can write news posts and assign a particular topic to the post.
- As a specialty (or, as one could say, as a consequence of incorrect design), table news_range is used to assign those news posts to either lectures or institutes. This is done using the attribute range_id, which does not constitute a real foreign key constraint, but is joined against seminar_ids and institut_ids, which are disjoint.
- Additionally, users can write comments concerning arbitrary lectures, which are then presented in the context of the respective event.
- Table seminar_user takes over two alternative functions: first of all, it is used to store information about users attending events. Second, this information is used for notification purposes: the attribute notification stores in coded form if users should be informed about updates of the event.
- Finally, table object_user_visits is merely used to make event handling possible: whenever a user of the system accesses any event, this visit is stored together with the kind of information he or she exactly accessed (forum posts, documents,...), and when the access happened (visit_date).

Most of the presented tables contain additional information about the creation date and the most recent modification date, stored in the attributes mkdate and chdate, which is also used to determine updates the users did not yet acknowledge.

Further Analysis Another important aspect of *Stud.IP* concerns its typical data access patterns. To verify the assumption that different phases in the system's operation lead to different access patterns, we analyzed the database logs to determine which tables are updated frequently or infrequently. To get representative results, updates were recorded during a whole week. The results have been purged by removing all modifications of tables that are merely needed for notification purposes.

We analyzed the update behaviour during the semester break and during the beginning of a new semester. Figure 2.4 represents the distribution of update, insert and delete statements during the semester break.

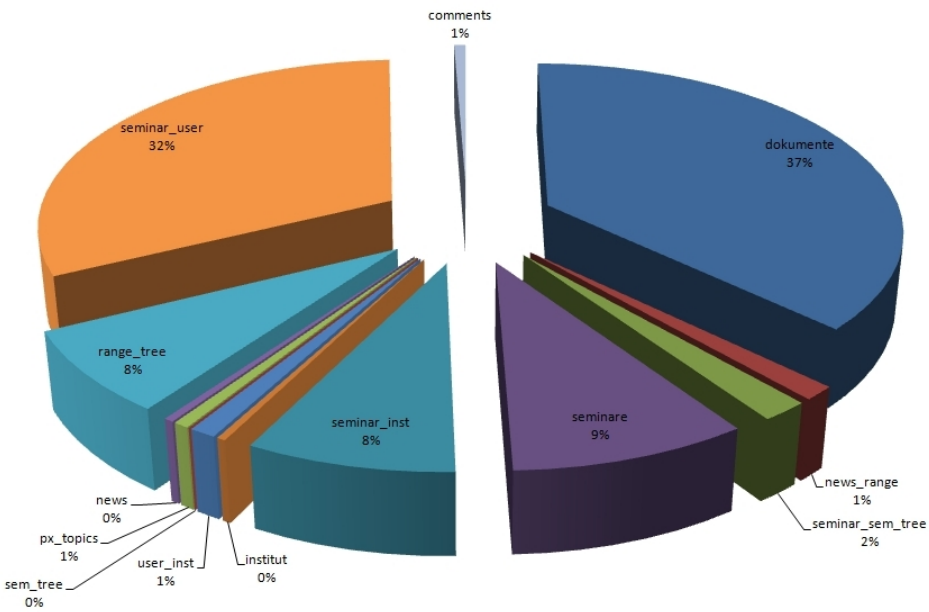


Figure 2.4: Update distribution during semester break

As one would assume, the results yield that during the semester break, updates mainly concern tables that represent administrative information like the assignment of students to courses (seminar_user), the organizational structure of the university (range_tree) or documents for upcoming events (dokumente). In contrast, information that is typically changed while a lecture takes place (like news, for instance) are seldomly updated during that phase. Figure 2.5 illustrates this aspect within the data model: dark coloured entities turned out to be updated very frequently, while in

contrast the brightly coloured tables tend to be very stable, i.e. they are subject to changes very seldomly.

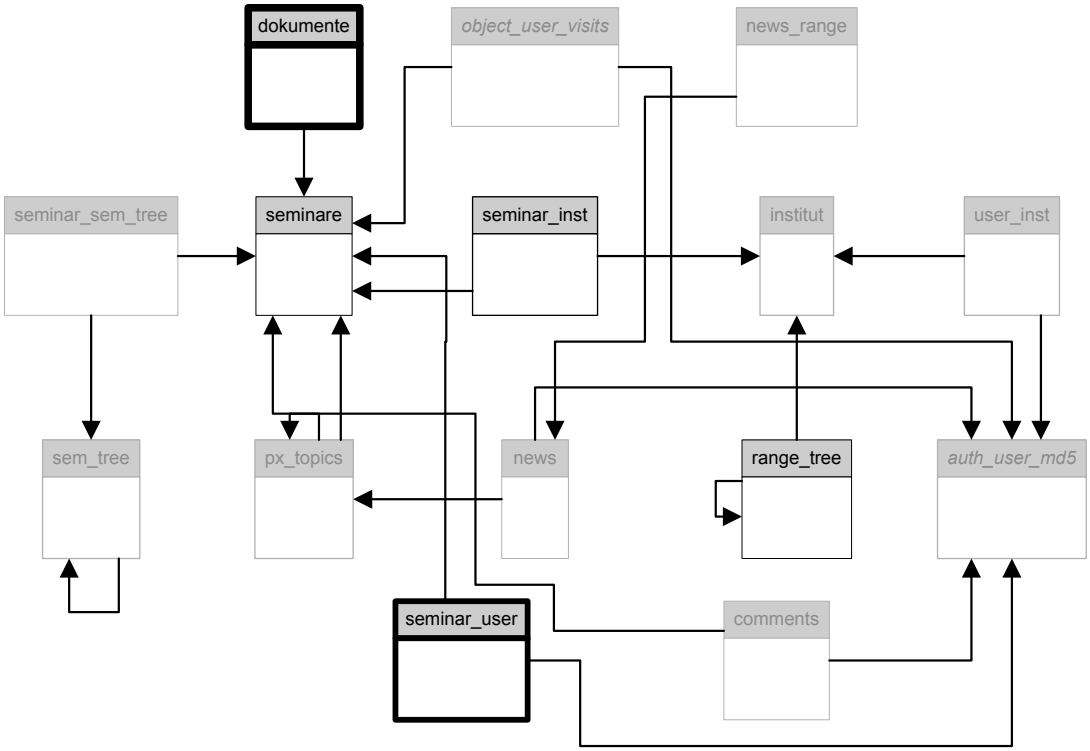


Figure 2.5: Write access during semester break

The same analysis has been repeated during the beginning of winter semester 2008/09: again, a week's logfiles were analyzed to obtain the distribution of updates over all relevant tables. Figure 2.6 reveals that the pattern of usage is significantly different.

The most obvious difference concerns table dokumente, which is updated much more frequently than during the semester break, which can easiliy be explained by lecturers and students working on their lectures and thus storing documents in *Stud.IP*. Second, many organizational tables, like seminar_user, range_tree, seminar_inst and seminare are modified by far less frequently, since these tables contain information that is usually maintained during semester breaks and then remains stable for the ongoing term. The different usage situation during the semester is displayed in figure 2.7.

Independent of the update probabilities, the number of instances for each entity (independent of the current phase) has been examined: we can observe that some of

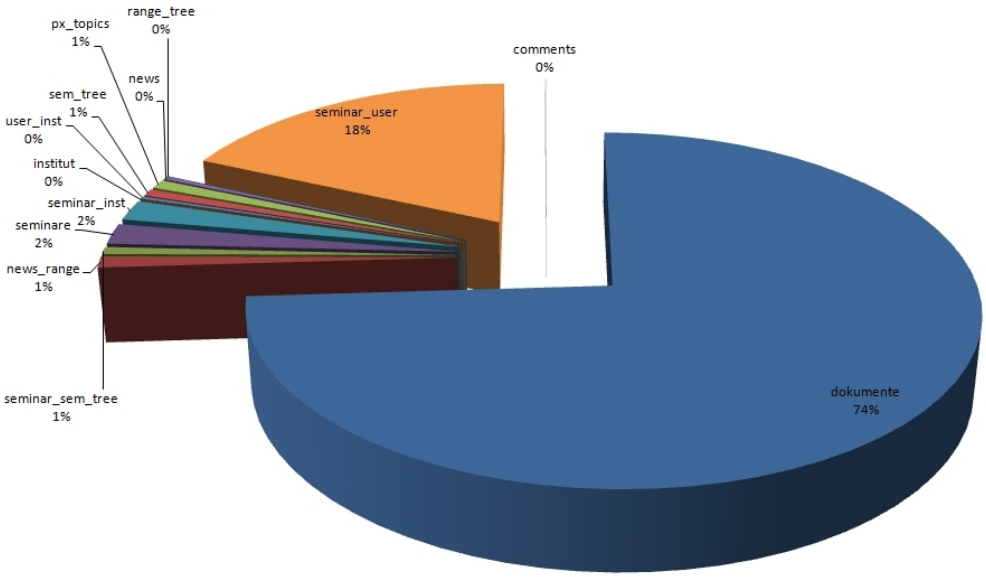


Figure 2.6: Update distribution during ongoing semester

the tables contain very few entries (institut, range_tree), while other tables like seminar_sem_tree, object_user_visits or dokumente contain significantly more tuples (cf. table 2.1). We also observe that the update probability and the amount of data per entity are not correlated to each other.

2.1.1.4 Essential Cognitions

Our analysis yielded various results: first, we discovered several deficiencies of the implementation. Beyond that, we also identified some interesting characteristics regarding a generic solution for the event-handling problem definition:

Drawbacks of Pull vs. Push Due to the implementation according to the pull-paradigm (i.e. during every user access it is checked whether data has been modified since the user's last access), many unnecessary read accesses to the database are executed. In addition, the pull-based approach (both during the notification batch run

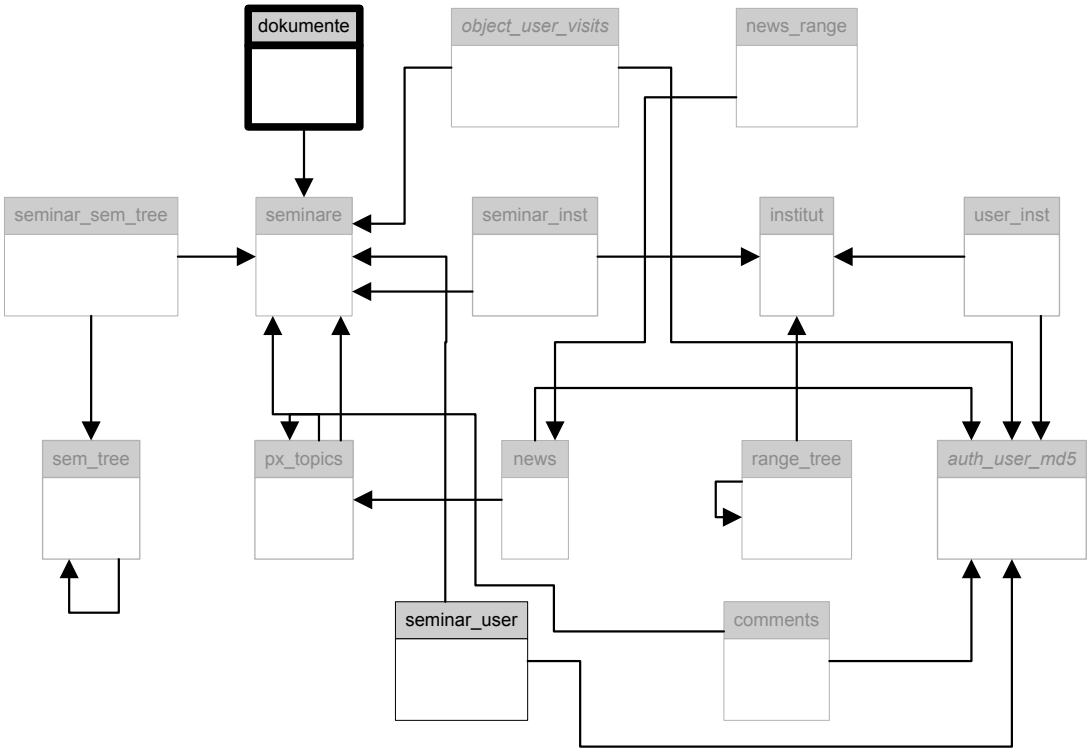


Figure 2.7: Write access during ongoing semester

and in the course of the portal implementation) does not satisfy the users' requirements of timely notifications. This could be resolved by using a push-based approach, reacting to updates immediately.

Missing Flexibility of the Implementation Regarding the common non-functional requirement "Anticipiation of Change", the analyzed solution fails completely: the implementation is completely hard-coded and tied to the underlying data model and thus very hard to maintain. In addition, the different parts of implementation are (due to the pull-paradigm) widely spread all over the source code, because updates have to be detected in many different contexts. This strong cohesion between different functionalities additionally hinders the system from being maintained, for instance by adding a new and previously unconsidered type of "data cloud".

Table	Number of tuples contained
auth_user_md5	11,685
comments	1
dokumente	31,937
institut	200
news	260
news_range	614
object_user_visits	1,821,147
px_topics	16,486
range_tree	199
seminare	5,802
seminar_inst	6,821
seminar_sem_tree	24,344
sem_tree	690
user_inst	12,397

Table 2.1: Number of tuples in selected tables of *Stud.IP*

Implicit Subscriptions Another discovery in the examined notification system is the fact that for several use cases, an implicit subscription can be derived from the associations between subscribers and subscribables. For instance, users are connected to their respective events via an $m : n$ association. Similarly, users are also related to the institutions from which they want to receive update notifications. We therefore argue that in many cases notification requirements can be directly derived from associations in the data model, which is what we call *implicit subscriptions*.

Explicit Subscriptions In contrast, users of *Stud.IP* want to be kept informed about updates of data entities which they are not related to at all. For instance, a student who is in the phase of preparing his semester schedule might select several upcoming events which he did not yet apply for, but nevertheless wants to be kept up-to-date about any changes. We call this *explicit subscriptions*.

Transitive Notifications An additional observation we made is that in several cases, the information system's users need to be notified about updates of entities that are indirectly connected to any other entitiy they subscribed to, be it explicitly or implicitly. As an example, consider the hierarchic structure of institutes: if one of the users subscribes to a top-level institution, he very likely also wants to be informed about updates of any subordinate institution, although he did not directly subscribe to the

respective entity. Another similar situation can be observed regarding documents and lectures: if some user subscribes to a lecture, he also wants to know about updates of any associated documents, even if he did not directly choose those documents as notification sources.

Coherence between Data Model and Notification Semantics Another meta-recognition can be derived from implicit subscriptions and transitive notifications: for almost every way along which update events should be brought from subscribables to subscribers, a corresponding association path exists in the information system's data model. Apart from explicit subscriptions, which can connect arbitrary subscribable entities and arbitrary subscribers, every notification rule somehow corresponds to the underlying data model.

Need for Attribute Monitoring From the use cases examined within *Stud.IP*, we can also derive another requirement: a generic event-handling system must be able to include and exclude individual attributes from the list of monitored attributes for any entity: Consider for instance the entity for documents in *Stud.IP*: this database table also contains an attribute storing the total number of downloads of this particular file, although an update of this value has no effect with respect to its subscribers, since a change of download frequency does not mean a semantic update of the document and is nothing a user wants to be informed about.

Distribution of Update Probability and Cardinality Another interesting characteristic, looking at the data model, is the heterogeneous distribution of cardinalities and update probabilities. Depending on the particular entity, both properties must be precisely examined since they strongly affect the way an event-handling implementation has to be designed.

Situation-Dependent Changes of Update Probabilities Finally, the comparison of update characteristics during the different phases "semester break" and "ongoing semester" reveals that, depending on the current situation of the system's use, update probabilities are subject to change. If an optimization strategy uses index structures to speed up the computation of queries because they assume that these index structures need to be updated seldomly (because of infrequent updates of the underlying data), then this optimizer must be able to adapt to constantly changing situations.

All these observations will later be used to determine the overall requirements for an event-handling system. Before that, we will present another system which we evaluated to get as general results as possible.

2.1.2 InfoWiss

The second software system we examined with respect to its notification facilities and requirements is the knowledge management system *InfoWiss* which was developed in the course of the author's diploma thesis [Gup01].

2.1.2.1 System Overview

InfoWiss was designed as a prototype of a corporate knowledge management system. Its main purpose is to store information and knowledge (documents, forum posts, competency profiles, ...) in a structured way. The core of *InfoWiss* is made up of a central multilingual taxonomy, which constitutes the organizational basis for information classification. Figure 2.8 shows a screenshot of the taxonomy modelling tool. The second building block of *InfoWiss* is a topic-based notification facility, which will be evaluated in detail in the following.

2.1.2.2 A Spotlight on InfoWiss' Event-Handling

In *InfoWiss*, the central taxonomy builds the basis for all subscriptions. Users are able to subscribe to any topic and - if desired - to all subtopics as well. Modifications of the taxonomy are automatically accounted for, i.e., if new subtopics are added to any topic *after* a user subscribed to it, these new subtopics are automatically subscribed to, too.

These subscriptions are evaluated whenever an information fragment (a document, a news posting, ...) is updated in *InfoWiss*: since each of those knowledge items has to be associated to at least one topic, thus classifying it, all users that subscribed to one of these topics are automatically informed about the update. This information is published to the users either per email (via a daily batch job) or on their individual portal page.

On the next pages, we will describe the realization of this notification component in detail.

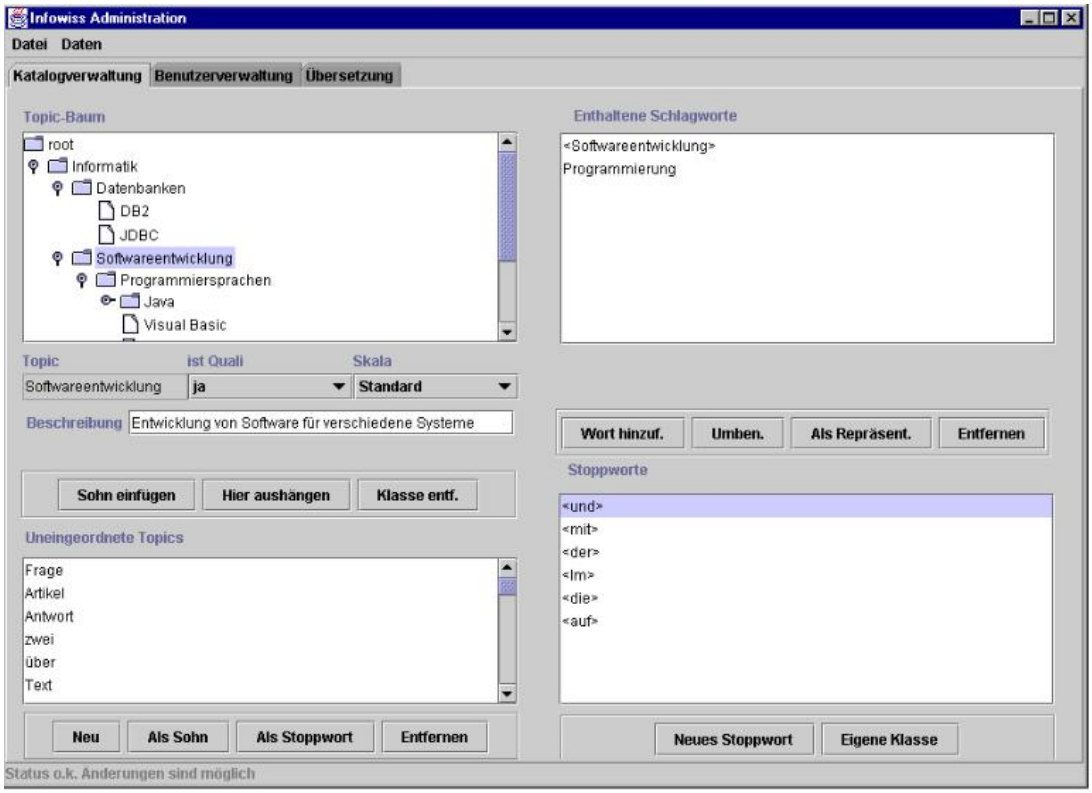


Figure 2.8: Screenshot: Taxonomy model in *InfoWiss*

2.1.2.3 Software-Technological Analysis

Again, we analyze the four aspects from section 1.1:

- Specification of data clouds
  Figure 2.9 illustrates the specification of data clouds in *InfoWiss*: by subscribing to topics from the central taxonomy, all adjacent documents are automatically declared to be part of the user's data cloud. Due to the generic implementation of information items, this subscription mechanism can be applied to any kind of stored knowledge.

- Determination of update events
  In contrast to *Stud.IP*, where the application is responsible for monitoring updates and creating the respective notifications, *InfoWiss* uses an active database

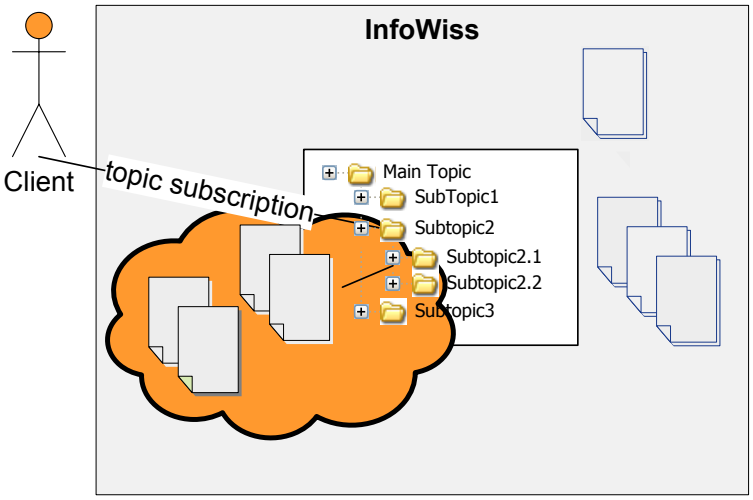


Figure 2.9: Specification of data clouds in *InfoWiss*

approach: all relevant database tables are monitored by triggers. Thus, any update of the data, independent of the source of the update, can be centrally monitored. Modifications of the application (new ways of entering data, modifying data from external systems via interfaces, ...) do not impose the need to care for event-handling but leave this task to the central data storage, i.e. the database.

- Matching update events
  Since the users' interest profiles are specified based on the taxonomy and all information entities are connected to at least one topic within that taxonomy, the matching between updated knowledge items and the respective subscribers can be performed by joining information entities, the taxonomy and users' profiles.

- Delivery of update events
  Similar to *Stud.IP*, *InfoWiss*' notification events are delivered to users via email or on each user's individual portal page. To send email notifications, a batch job has to be scheduled which regularly checks for new notification entries and informs the corresponding users.

Data Model Analysis To gain more insight into the realization of *InfoWiss*' notification functionality, the data model, which is presented in simplified form in figure

2.10, has to be analyzed. Again, coloured tables are mainly used for event-handling purposes.

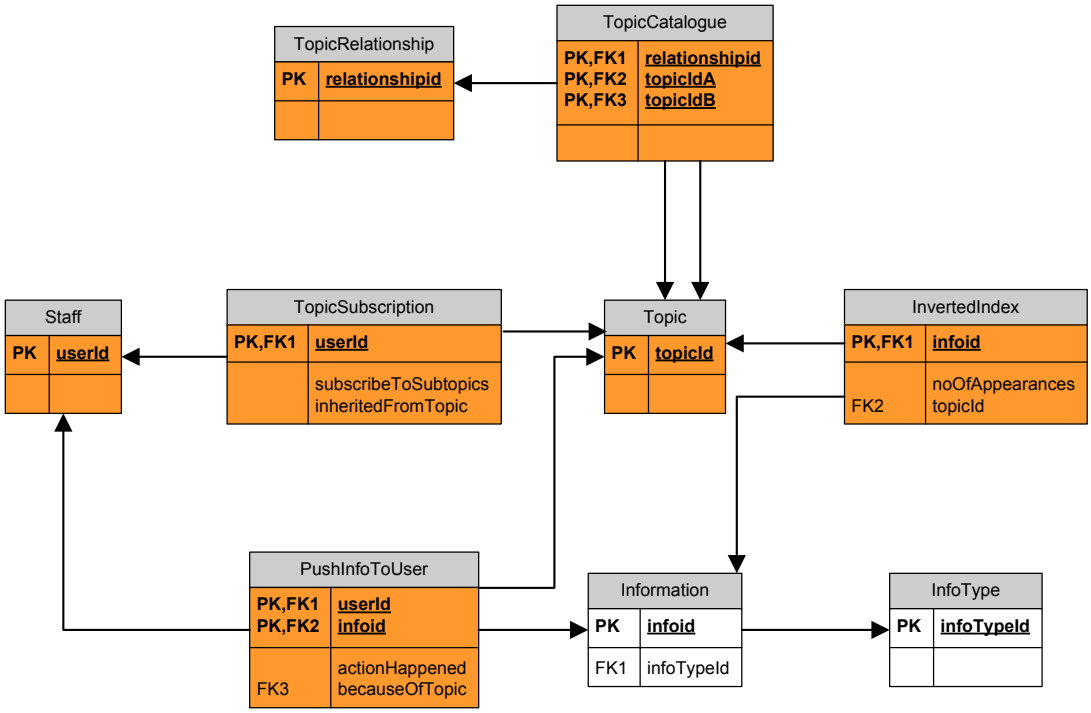


Figure 2.10: Excerpt from *InfoWiss* data model

The following tables constitute the foundation of *InfoWiss*:

- Table Topic is used to store all topics that are part of the taxonomy. In *InfoWiss*, topics are only identified by a unique id, while all terms (i.e. the words) that represent that particular topic are stored in a dedicated multilingual table, which is, however, not in the focus of this analysis.
- To build a taxonomy, these topics have to be put in relationship to each other. The different types of relationships that can be used ("is-a", "is opposite of", ...) are maintained using table TopicRelationship.
- Finally, the taxonomy is completed by associating any two topics with each other, represented by an entity of TopicCatalogue. By referencing the respective TopicRelationship, the type of the relationship is specified.

- Any kind of information that can be stored in *InfoWiss* is kept in table Information. There are several different subclasses of Information for different types of knowledge (documents, forum posts, ...) which are not relevant to our analysis.
- To represent the different subclasses, table InfoType is used, which is a standard approach to model inheritance in relational databases.
- To assign any topic to any bit of information, an InvertedIndex is used. This index is built by (manually or automatically) extracting keywords from the different information fragments, thus classifying the information. The index can be queried to either determine all topics for a particular document or to find all bits of information concerning a particular topic.
- Table Staff stores information identifying every user of the system, i.e. maintaining the list of all possible subscribers within *InfoWiss*.
- As mentioned before, *InfoWiss* uses the subscriptions of users to their topics of interest. The actual subscriptions are stored in table TopicSubscription. This table contains a specialty: since users can subscribe to a particular topic *and* all of its subtopics, the attributes subscribeToSubtopics and inheritedFromTopic are necessary. subscribeToSubtopics stores a boolean value specifying whether the subscription is to be valid for all subtopics. This information is then propagated to all subtopics, so that all inherited subscriptions are materialized, referencing the originating topic in attribute inheritedFromTopic.
- As a final component, table PushInfoToUser contains all notification events which have to be sent to users. Besides the updated document, additional information about the update event, together with information about the topic that was relevant for the subscription, is stored.

Further Analysis Based on this insight into the data model semantics, we will further analyze the realization of the event-handling system in *InfoWiss* in the following.

One of the details of *InfoWiss* that are worth mentioning is the overall design, choosing to use the database as the central point of update detection. All updates of data are monitored centrally using standard database triggers, thus realizing a real push-mechanism triggered by the modifications of data. In contrast to the realization within *Stud.IP*, which is closely tied to the application, this approach offers several advantages:

- The event-handling mechanism has to be realized only once; no matter which components of the software modify data, the cross-cutting concern of subscription and notification management does not have to be realized separately.
- Additionally, any third-party application working with this database can automatically profit from the event-handling functionality by simply updating data in the central storage.
- Finally, any modifications of the event-handling semantics (for instance, different interpretations of topic subscriptions) have to be realized only *once*, instead of having to modify a multitude of spots in the application's code.

Another interesting aspect is the realization of subtopic-subscriptions. Basically, there are two possibilities to do so: first, after every update of an arbitrary bit of information, all subtrees of any topics assigned to this information could be checked for potential subscribers. The diametral solution would be to pre-compute all transitively subscribed subtopics and store the complete list in the users' subscriptions profiles. *InfoWiss* makes use of the fact that both the central taxonomy as well as the users' subscriptions are updated rather seldomly, while in contrast different information fragments are updated frequently. Based on this assumption and considering runtime performance, it is obviously better to determine all descendants of a subscribed topic and query this precomputed list instead of traversing the taxonomy after every monitored update. However, this causes significant realization overhead, since changes of a user's subscription profile and/or of the taxonomy can make it necessary to recompute the subtopic lists. Thus, *InfoWiss* uses designated database triggers to maintain the subtopic subscription lists and keep them up-to-date after every modification of the taxonomy.

2.1.2.4 Essential Cognitions

Similar to the analysis of *Stud.IP*, we can summarize our results with respect to the following aspects:

Advantage of Push over Pull Due to the trigger-based realization, every data modification can immediately be transfered into the corresponding notification. Although *InfoWiss* simply stores the notification entry in the database for later use, one could easily extend the system to generate emails or react timely in any other appropriate manner.

Central Monitoring of Updates in Database The central trigger-based realization of the event-handling component within the database makes it easy to extend the system with respect to future requirements. The realization is *not* spread all over the application and can easily and centrally be maintained and modified.

Realization of Explicit Subscriptions using Implicit Subscriptions In *InfoWiss*, explicit subscriptions of particular documents, posts, ... are not possible. Instead, all subscriptions have to refer to one of the taxonomy's topics. However, the subscription to these topics is realized using the association TopicSubscription. This can also be considered an implicit subscription, since every user who is related to one of the topics is automatically informed in case anything related to that topic is updated.

Transitive Event Propagation As already discovered during the analysis of *Stud.IP*, transitive event propagation can also be found in *InfoWiss*' event-handling system: whenever a piece of information is updated, this update information has to be propagated to all associated topics and, transitively, to all of its subtopics, so that the implicit subscribers of this topic (i.e. the users) can be notified about the update.

Hierarchical Structures Again, hierarchical structures can be identified: in this case, the taxonomy constitutes the only, but central, hierarchy in the data model.

No Limit of Propagation Depth In *InfoWiss*, event propagation is not limited in its range. This means that an update of a document related to a topic that is on top of the hierarchy would be propagated through to all topics in the sub-tree below the updated topic, no matter how deep the taxonomy hierarchy is. This might be considered a deficiency, since on the one side it may lead to performance issues in case of large taxonomies, and on the other side the semantic relevance of such a root update may not be significant enough for a subscriber of one of the leaf topics to be informed.

Only one Direction of Propagation Another drawback of *InfoWiss*' event propagation system is the fact that it is only possible to propagate updates from supertopics to subtopics, but not vice versa. In an ideal system, it should be possible to propagate updates along all kinds of associations (e.g. also from topics to their counter topics, if there is such an association type) and in any desired direction (e.g. also from subtopics to their supertopics).

Precomputation of Propagations Finally, the precomputation of subtopic lists to allow faster queries represents an important aspect: whenever event propagations are possible within data structures that seldomly change but are frequently queried, an ideal event-handling system should be able to precompute most of its propagation data, rather than evaluating the whole propagation path at query-time.

2.1.3 Further Use Cases in Brief

In the following, we briefly list several additional use cases and existing systems and point out the respective notification aspect.

Document Management Systems Most document management systems are used as a tool to collaborately manage company-wide documents. These documents are usually stored in a hierarchical folder system. To keep users aware of modifications to documents made by their co-workers, these systems, for instance Intland Codebeamer [Int] with its integrated team collaboration features, often offer functionality to subscribe to individual documents or folders. Similar to the subscription mechanism of *InfoWiss*, the ability to subscribe to all subfolders of a given folder is also a desired feature.

Metadata-Based Archive Systems A more generic approach to the storage, administration and retrieval of information are metadata-based archive systems. These systems are typically able to maintain a catalog of possible metadata for specific documents and allow the users of these systems to store and retrieve documents, classified according to those metadata schemata. A very sophisticated instance of such a system is the german remote sensing data center's multi-mission ground segment Data Information and Management System (DIMS) [Kie02] [KF07], which is based on a self-descriptive metadata model. Those systems require notification functionality on two layers: first and foremost, users want to be able to subscribe to all documents being elements of the result set of particular metadata queries. Further on, administrators could also want to be informed about modifications of the metadata schemata to stay up-to-date with all modifications of the datamodel and thus be able to maintain a consistent and redundancy free metadata catalogue.

Skill Management Systems Many companies support their human resources departments by introducing skill management systems, storing qualification profiles of their

employees. Such systems also offer a lot of potential with respect to event-handling: team leaders could be informed about new qualifications acquired by their team members, while management and specialists might by interested in any new colleague who gained knowledge in a field they are working on themselves.

Project Management Tools Similar situations and use cases can be observed in the field of project management systems. Project managers most likely want to be automatically informed about any updates of the project state, the completion of their team members' tasks, unforeseen problems that are recorded in the project management system, etc. Similarly, team members could profit from an immediate notification about changes of their tasks, updates of the schedule, etc.

This list could be extended almost infinitely: from bug tracking systems over web content management systems to stock-trading systems, nearly every computer based information system could profit from the introduction of event-handling functionality. Since the requirements are almost the same for every kind of such a system, we will abstract from the concrete field of application and subsume those requirements, based on the insights we gained from analyzing the above-mentioned examples. These requirements are listed on the following pages.

2.2 Inferred Requirements

As a conclusion of the analysis presented in this chapter, we identified a list of requirements that have to be fulfilled by an ideal event-handling system. We classify these requirements into three categories: semantic requirements, prescribing *what* the solution should be capable of, technical requirements, specifying *how* this functionality has to be realized technically, and requirements considering software engineering, demanding how developers and maintainers of an event-handling system should be supported to simplify their work.

2.2.1 Semantic Requirements

First of all, we can reduce the semantic requirements that have to be fulfilled by the event-handling system we want to design to the following requirements:

/R1.1/ Entities must be markable as subscribable It must be guaranteed that every entity class that is stored in the underlying information system can be marked as *subscribable*, i.e. as an entity class that has to be monitored. An update of an instance of this class must trigger the event processing.

/R1.2/ Entities must be markable as subscriber As a counterpart to *subscribables*, every entity class must also be markable as a *subscriber*. Every instance of this class can then be a potential receiver of update events.

/R1.3/ Monitoring of subscribables must be limitable to individual attributes It must be possible to tag individual attributes of a subscribable entity as *observed*. Only updates of these particular attributes may trigger the event processing, thus avoiding reactions to meaningless update events.

/R1.4/ Implicit subscriptions must be supported It must be possible to specify that relationships between subscribables and subscribers designate an implicit subscription. In that case, any detected update of the participating subscribable must lead to a notification of the corresponding subscriber.

/R1.5/ Explicit subscriptions must be supported At runtime, it must be possible to maintain a list of explicit subscriptions between subscribers and subscribables, independent of any associations between them. Every entry in this list must lead to the delivery of a notification to the subscribers in case the corresponding subscribable has been updated.

/R1.6/ Transitive propagation of update events must be supported It must be possible to specify that associations between two subscribables lead to a transitive propagation of update events along this association. If an instance of the originating subscribable is updated, all instances that are adjacent to this instance via such an event-propagating association must be considered as updated, too.

/R1.7/ Impact of transitive propagation must be limitable Further it must be possible to limit the impact of a transitive event-propagating association, especially in case of reflexive associations. By limiting the impact to a maximum number of "hops" along the association, the impact of an update can be reduced.

/R1.8/ Transitive propagation along associations must be directed It must be possible to specify the direction of event-propagating associations, i.e. the source and the target of the propagation must be clearly defined.

As we were able to demonstrate during the analysis of *Stud.IP* and *InfoWiss* (and as we will verify based on other real-life use cases in part IV), almost any event-handling use case can be assembled using the constructs described above.

2.2.2 Technical Requirements

Independent of the semantic demands, several requirements concerning the technical realization can be postulated:

/R2.1/ Updates must be monitored and handled centrally in data storages Instead of realizing the update monitors across the whole application, the central data storage (e.g. relational database, XML database, flat file, ...) must be monitored for changes.[2]

/R2.2/ Updates must be actively detected and pushed to subscribers Any updates in the central data storage must be detected immediately (i.e., the event processing must be actively triggered by the update itself), instead of regularily polling for updates using a time stamp or anything similar.

/R2.3/ All event-handling constructs must be based on the system's data model The semantic constructs introduced in section 2.2.1 must be applicable as an extension of a structured model of the information system, e.g. an ER diagram, UML model, XML Schema, etc.

/R2.4/ Hierarchical structures must be supported efficiently Hierarchical data structures, which mostly manifest themselves in the data model in the form of reflexive associations, must efficiently be supported, especially regarding event propagation.

/R2.5/ Precomputation of event propagation has to be used, where appropriate To efficiently handle large and tightly interwoven data structures, the event-handling

[2] We do not consider distributed data storage systems in our work. However, we will give a very brief outline how our approach could be applied to distributed database systems in chapter 12.

system must be able to precompute the impact of event propagation. The system has to determine where this precomputation is appropriate, by estimating or analyzing update behaviour and usage characteristics.

/R2.6/ The system has to adapt to different usage characteristics during lifetime Since update behaviour and usage characteristics tend to change over time, the system must be able to adapt itself to these changes dynamically and - where appropriate - change the precomputation strategies, if necessary.

2.2.3 Software-Engineering Requirements

Although the technical requirements specify the system characteristics that the final implementation of the event-handling component has to provide, they do not prescribe how developers have to be supported in building and maintaining such a system. These maintenance requirements are particularly important, since, according to common knowledge, two thirds of the effort that are put into a software system are required for its maintenance, while only one third goes into development.

/R3.1/ Specification of event-handling semantics has to be declarative The fundamental event-handling semantics, based on the above-mentioned constructs, must be applicable in a declarative way. This simplifies development, maintenance and modifications of an event-handling system and increases flexibility.

/R3.2/ The system must be open to future modifications The event-handling framework must support modifications of an in-force event-handling component. Both changes in the semantic specification as well as changes of the underlying information system must be manageable.

/R3.3/ The semantics must be open to future modifications It must be possible to change the semantics of an already specified event-handling component without having to adapt the underlying information system (of course, this is only possible if the semantic changes are compatible to the previous version). As an example, consider the introduction of a new semantic construct or a change in the interpretation of implicit subscriptions.

/R3.4/ Concept must be applicable to existing systems in a non-invasive way Finally, it must be possible to develop an event-handling component on top of an already existing information system, without having to modify the legacy system.

If all these requirements are fulfilled, we can claim that the proposed solution is *adequate*, *technically mature*, *easy to build*, *non invasive* and *maintainable with minimum effort* - i.e., our general requirements are fulfilled.

2.3 Summary

In this chapter, we detailed the problem statement "How to enhance information systems with event-handling in a non-invasive way". We did so by taking a detailed look at two information systems that support event-handling in a basic, but state-of-the-art way. Based on this analysis, we identified their weaknesses and the universally valid characteristics of event-handling components. Finally, we unified all these recognitions and derived a list of semantical, technical and software-technological requirements, which will be the basis of the work presented in this dissertation.

"Accept good advice gracefully – as long as it doesn't interfere with what you intended to do in the first place."

Gene Brown

3 Existing Approaches

The following chapter takes a closer look at existing approaches in the field of event-handling systems, also known as publish/subscribe paradigm. To classify the systems, we first present an architectural classification scheme. Based on this scheme, we will enumerate several existing concepts, solutions and software systems that handle one or more of the architectural aspects of an event-handling system. These approaches will be evaluated with respect to the requirements we postulated in chapter 2. As it will become clear, none of the existing approaches is able to address all demands, so that a new approach to this problem has to be developed.

3.1 Classification Scheme for Publish/Subscribe Systems

Publish/subscribe systems are a common technique used to couple different systems and/or users by providing a possibility to inform arbitrary parties about updates that take place in a different system and/or by a different party. Subscribers have the ability to express their interest in a type of event, or a pattern of event types, and are subsequently notified about any event, generated by a publisher, which matches their registered interest. An event is then asynchronously propagated to all subscribers who registered their interest in that given kind of event. The participating systems and

parties are usually decoupled in three dimensions: space, time and synchronization [EFGK03]. A simple publish/subscribe system can thus be divided into the following components, as figure 3.1 shows:

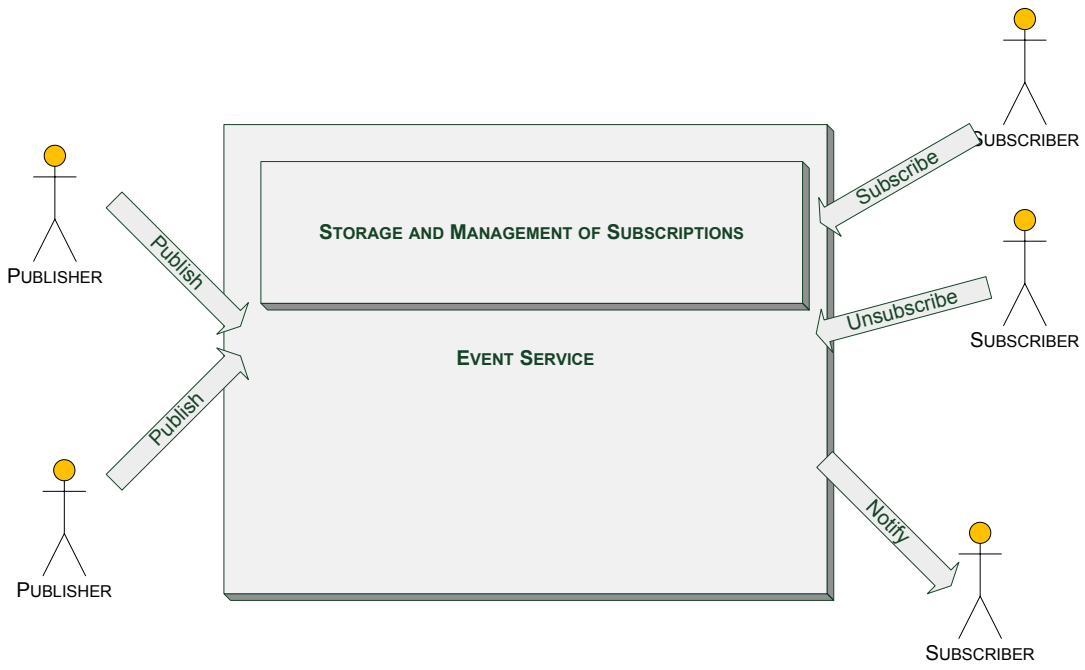


Figure 3.1: Generic publish/subscribe architecture by Eugster et. al. [EFGK03]

- *Publishers*, who advertise arbitrary events (e.g. the modification of data) to a central event service which receives those events
- *Subscribers*, who, as a counterpart to the publishers, signal their interest in particular events by subscribing to the central event service. If these subscribers lose their interest, they can undo their subscription by unsubscribing again
- A central *event service*, storing all subscriptions, accepting the notifications, determining which subscribers to notify and finally notifying all relevant subscribers about the particular events

Instead of going into more detail about these components, we will first refine the architectural overview and tailor it to the needs of our use cases.

The use cases we focus on are characterized by the following additional properties:

- Rather than developing a generic publish/subscribe system which is able to integrate and couple different heterogeneous software systems, we focus on an event-handling functionality within one particular application.
- Further on, we assume that all data that can be subject to change is stored within one (integrated) central data storage.
- As a consequence, subscriptions and notifications are also stored within the data storage.
- Another characteristic of our approach distinguishing it from traditional publish/subscribe systems is the way updates are advertised: in publish/subscribe systems, publishers usually classify the events they want to trigger, i.e. they have to have knowledge about the semantic meaning of "their" event. In our scenario, the classification of data updates has to be done by an application component. All updates are performed on the application data, so that the central event matching can be performed there. In particular, subscription semantics are thereby made context sensitive.
- Before events can be matched, they have to be detected, i.e. the data storage has to be monitored for any changes that could potentially lead to further event processing.
- Finally, the matching of events has to be performed by comparing the detected update events to the subscription specifications, i.e. to the interest of the system's subscribers.

In addition to the functional aspects discussed above, the following software technological requirements find their way into the architecture:

- As we demand that our approach has to be usable in a declarative way, we need a description language for all subscriptions. In our approach, a meta-model has to be developed that can be used as the specification language.
- Based on the meta-model, transformations rendering the model into an executable form that suits the particular application system are necessary.
- The desired solution not only has to be able to match detected events against the different subscription specifications, but it also has to be prepared for high throughput and work efficiently. Thus, an optimization component that decides how to optimally perform the matching has to be part of the architecture, too.

- An often forgotten aspect is the security of the event-handling system: since most information systems limit the direct and explicit access to the data (for instance by assigning roles and rights), it must be assured that no one can access data indirectly by placing subscriptions which would lead to notifications that contain information which is usually hidden from the respective users. Thus, security mechanisms have to be applied to both the subscription and the notification delivery component. In this dissertation, we will however not deal with this aspect.

All these preconditions, assumptions and requirements result in the architectural classification scheme shown in figure 3.2.

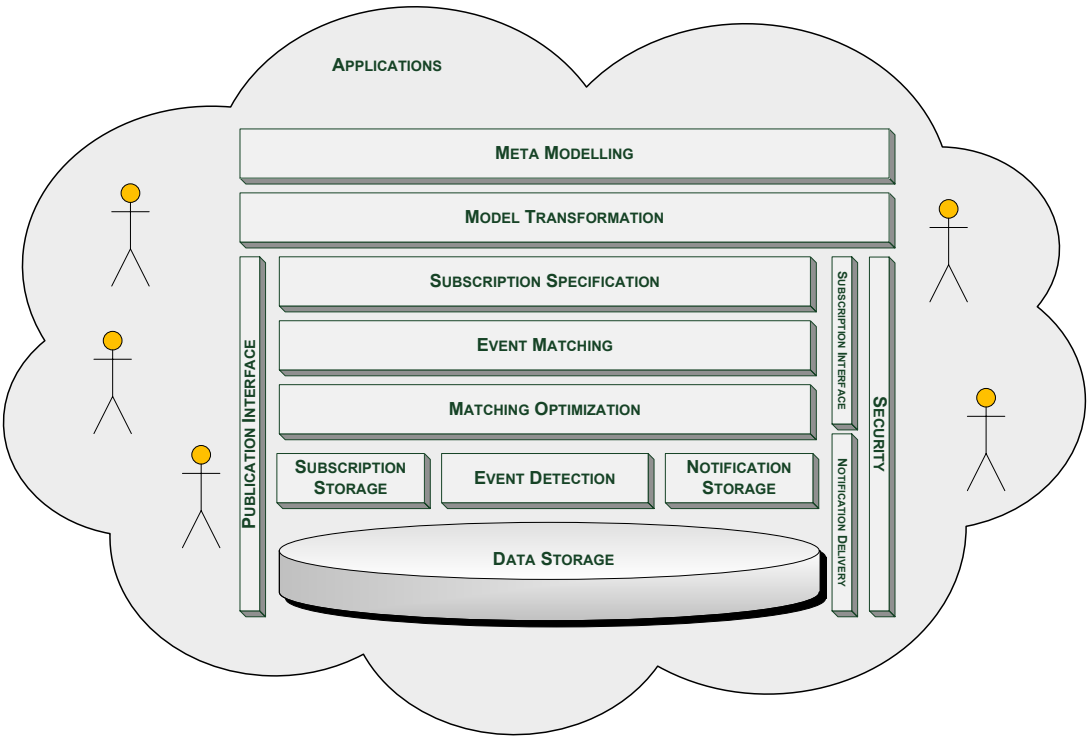


Figure 3.2: Detailed architecture overview

This architecture, which follows the modular architecture proposal by Filho et. al. [FdSR03], contains the following components:

Applications As motivated above, we look at event-handling functionality that can be used within different kinds of applications. We thus have to embed the overall architecture into the respective application, i.e. the corresponding information system.

Meta-Modelling The above-mentioned meta-model (and the respective modelling tools) to design the event-handling semantics constitute the top level of our architecture. This layer also contains a precise, formal definition of the meta-model's semantics.

Model Transformation Since the generic meta-model has to be independent of the actual implementation of the underlying event-handling system, corresponding model transformations are needed. The transformations take the formal event-handling specification as an input and transfer it into the target system environment.

Subscription Specification Primary target of the model transformations is the subscription specification. This layer contains the system-specific realization of the notification semantics, i.e. the rules whom to notify about which updates.

Event Matching An important part of the architecture is the layer which matches detected events against the subscription specification. There is a variety of ways how this matching can be performed, as we will see in the following overview of related work.

Matching Optimization Finding the most efficient way of performing the matching procedure is part of this architectural layer. The underlying concept, which will be detailed in chapter 7, is applicable to any kind of implementation. However, the concrete implementation of the optimizer has to be implementation specific.

Event Detection The detection of events is one of the most crucial parts of the architecture. Depending on the type of data storage, all relevant modifications of the data have to be monitored and propagated to the matching layer.

Data Source The central data storage (as we have mentioned previously, distributed data storage systems are not in the focus of our work) contains all information that

has to be monitored for modifications. The overall architecture does not depend on a particular type of storage, i.e. the concept is applicable to relational databases, XML databases, flat files, etc.

Subscription Storage This layer cares for the storage of subscription information (i.e. the subscribers' interest). In our scenario, this storage is integrated into the central data storage, i.e., subscription information is stored together with payload data.

Notification Storage Likewise, notifications, which have not necessarily been post-processed immediately after they were created, have to be stored in the central data storage, which is done by the notification storage layer.

Publication Interface A lateral layer contains the publication interface: event producers, i.e., in our scenario, updaters of data, need an interface to advertise their events.

Subscription Interface The mirror image of the publication interface is the subscription interface: subscribers' interest has to be announced to the event-handling system by subscribing and unsubscribing, similar to the generic classification by Eugster et. al. [EFGK03].

The following two architectural layers are not in the focus of our work. However, for the sake of completeness, they have to be mentioned, too.

Notification Delivery As soon as events have been detected and matched against all known subscriptions, the corresponding subscribers have to be notified. Thus, an important part of publish/subscribe systems is how to deliver those notifications to the (usually spacially and temporally separated) subscribers.

Security Finally, security is an important issue: this layer has to assure that subscriptions and notifications can not break the information system's access restrictions.

Although all these layers basically work independently of each other, supporting a strong separation of concerns, there are several building blocks that have to be coordinated: *Subscription Specification*, *Event Matching*, *Matching Optimization* and *Event Detection* make up the central block that is coupled tighter to each other than to

the surrounding layers. In addition, *Meta-Modelling* and the respective *Model Transformations* also have to be coupled. Finally, *Subscription Storage* and *Notification Storage* both have to refer to the same data and thus are usually realized in a similar way.

Starting from these building blocks of the architecture, we will present a survey of existing approaches that cover one or more of these layers. They will be classified by naming the role they play in our reference architecture and briefly evaluated against the requirements we postulated for our given use cases in section 2.2.

3.2 Overview of Existing Approaches

All approaches we examined are presented on the following pages, distinguishing between research (which usually focuses on a very particular aspect) and existing software systems, usually covering many of the different architectural layers.[1]

3.2.1 Research Projects

There exists a magnitude of research publications in the field of publish/subscribe systems and event-based systems. In the following, we will try to give an overview, ordered by the different architectural aspects the publications deal with. For a comprehensive paper on architectural aspects of building a publish/susbscribe system, cf. Fiege et. al. [FMG02].

3.2.1.1 Applications

A short list of use cases for event based systems has already been presented in chapter 2. In addition, literature offers many further use cases: generally speaking, the monitoring of arbitrary web pages and the possibility to get notified about interesting updates is an important scenario [PFLS00, PFL+00, RW97, FFM01]. Further on, groupware systems supporting the collaborative work of several users are an interesting field of application, too [HMJ+96, Rau96, KPdLB03], in particular if users modify a common set of documents [Thu00, DB92] or share a set of bookmarks that might be interesting

[1] Even if many of the presented research solutions may have found their way into "real" software systems, we decided to keep this differentiation for better comparability.

to other groups of users [LVA+99]. Further well-known fields of applications are digital libraries [BF06, FRS06] or web communities [FRS04].

A completely different field of application are location based services: both updates of geographic information as well as updates of users' locations can be subject to monitoring and cause the triggering of notification events [CMD02, MG02]. This field of application is even more important if it is applied to mobile devices [Zei04].

From an industrial view, events considering the movement of real-world entities, detected using RFIDs, have also been handled using event-handling systems [RJK+05].

The overall importance and up-to-dateness of publish/subscribe applications is also proved by the interest of the scientific community in tutorials about this topic, such as e.g. at the *VLDB '05* [IK05] or at the *SIGMOD '07* [CG07].

3.2.1.2 Meta-Modelling and Model Transformation

The usage of modern meta-modelling techniques like Model Driven Design (MDD) or Model Driven Architecture (MDA) has not yet extensively been applied to the field of publish/subscribe systems. Thus, to our knowlege, there exists only a single approach by Edwards et. al., who use model driven architecture for the high-level specification of configurations for publish/subscribe systems [EDS+04]. However, this approach does not present a solution of how to specify event-handling semantics based on the data model of an application, as our solution does. Furthermore, Edwards et. al. mainly focus on a model driven generation of configuration files and code fragments that can be used to publish and transport events to different kinds of subscribers, but the key question of how to detect those events in the publishing system, one of the core contributions of our solution, is not part of their work.

Although the meta-model developed by Jun Wang [Wan08] has been designed for the specification of event-based systems, his work focuses on the description of events that are provided and consumed by components in a distributed software system. He does not handle the aspect of how to describe the detection of relevant updates that cause the events, which is what we are going to do in our work by proposing an adequate event-handling concept, appropriate constructs based on the application's data model and the corresponding semantics.

3.2.1.3 Subscription Specification

The way subscribers specify their interest in events (located in the *subscription specification* layer, according to our architecture) constitutes a fundamental part of publish/-subscribe systems. Eugster et. al. differentiate between three different approaches: topic-based subscriptions, content-based subscriptions and type-based subscriptions [EFGK03].

Topic-based subscriptions rely on the notion of topics, which are usually identified by keywords. Subscribers specify their interest using those keywords, while publishers tag their events with the respective keywords. A matching between subscribers and publishers can thus be easily obtained by comparing the keywords of events and subscriptions to each other.

A more sophisticated approach are content-based subscriptions. Instead of tagging events with keywords, the content of the events themselves (attributes or meta-data of the event) is evaluated, leading to a classification of the events. Consumers then specify their interest by giving conditions (called filters) which the particular events must fulfil in order to be considered as interesting. Obviously, content-based subscription implies topic-based subscription, since topics can be modeled as part of the events' content [ASS+99].

Finally, type-based subscription filters events according to their type. Instead of considering the content of an event, clients can subscribe to different kinds of events (e.g. "StockQuotes" or "StockRequests"). These types are usually organized in taxonomic structures, so that subscriptions can be expressed referencing different parts of the type hierarchy.

For a detailed overview of publish/subscribe systems, see Mühl's work "Large-Scale Content-Based Publish/Subscribe Systems" [Müh02] and "Distributed Event-Based Systems" [MFP06] or, for recent work, the master's thesis by Jun Wang [Wan08].

In this dissertation, we will present the concept of how to specify content-based subscriptions. The novelty of our approach lies in the fact that we will be using the application's data model as a basis and provide appropriate yet simple constructs which can be used to specify the subscription semantics - an approach that has not yet been published previously.

Another part of the subscription specification is the specification of events themselves: a formal concept, based on an algebra, can for instance be found in Hinze's work [HV02a, HV02b]. Early approaches also used temporal algebrae for the subscription

specification languages, for instance Zhang and Unger [ZU96]. Again, none of these approaches has been designed to be directly applicable to models of the information system's data, so that our solution breaks new ground in the field of subscription specification.

3.2.1.4 Event Matching

The efficient matching of events against subscriptions has been in the focus of several research projects. A good overview of this topic has been published by Fabret et. al. [FJL+01].

One of the first publish/subscribe systems was developed by Krishnamurthy and Rosenblum, called *Yeast*: based on simple event patterns, subscriptions and the corresponding actions can be defined and are evaluated by a polling server process in order to notifiy the respective subscribers [KR95]. Obviously, this pull-paradigm based proposal does not fulfil our requirement to *actively* determine relevant updates.

Later, a more sophisticated matching algorithm for content-based subscriptions has been proposed by Aguilera et. al. [ASS+99]. By precomputing a decision tree, which is later used to decide whether an event matches a subscription, efficient matching can be guaranteed. Using a system of distributed brokers which keep a local copy of so called distributed hash tables, Tam et. al. showed that parallelization of the event matching and thus an efficient processing of events can also be reached in distributed environments [TAJ04]. Both publications provide a solution to the efficient matching of subscriptions with publications; however, they do not provide an integrated solution tailored to the needs that arise when enhancing "traditional" information systems with adequate event-handling functionalities.

Another research project focusing on content-based subscriptions is *Elvin4* [SAB+00], which provides a software library (implemented in *C*) to embed event-handling functionality into various applications. However, this approach is neither of declarative nature, nor does it consider the fact that much of the event-handling semantics can be derived from (or at least based on) information that is already present in the data model of the information system, so this is where our approach offers significant advantages when realizing use cases like the ones we have taken into account.

A different approach is followed by *Siena*, which uses a formal model of events, advertisments (= publications) and filters (= subscriptions) for the delivery of events to their destinations [CRW98]. Such a model has also been developed by Wang et. al., specialized on ontology-based publish/subscribe systems [WJL04]. Another ontol-

ogy based publish/subscribe-middleware, called *CREAM*, serves to integrate heterogeneous information systems [CBB03]. Again, like most of the published approaches to subscription matching, none of these solutions provides an integrated approach for use cases in which relevant data updates in an information system have to be determined and provided to the subscribers by considering relationships within the data model.

In a distributed, integrating approach, it is also important to know how to match and combine events from different sources. A solution to this problem, called composite events, has been proposed by Pietzuch and Shand [PS02], similar to the previously published concept of compound patterns, as they are used in *Ready* [GKP99]. This notion of composite events has later been rediscovered in the application field of XML databases, where Bernauer et. al. extended the event-handling platform *Snoop* [BKK04]. Similarly, Tian et. al. proposed a solution how to match XML document publications by using relational databases [TRP+04]. In addition, Hong et. al. evaluated how to support publish/subscribe over XML streams [HDG+07]. In an object-oriented context, Eugster and Guerraoui proposed how to implement a publish/subscribe system on top of Java, using structural reflection in order to evaluate and match events [EG01].

Further optimization potential can be exploited by grouping different topics, which the users can subscribe to, into virtual topic-clusters, as prototypically realized in the *Tamara* publish/subscribe system [MZV07]. A similar approach has also been developed by Zhang and Hu [ZH05].

All of the listed solutions can be considered as a reasonable extension of an event-handling system or as a specialized solution to particular sub-tasks, but do not handle the central question of how to specify relevant updates in structured information systems in a declarative manner.

3.2.1.5 Matching Optimization

The architectural layer of *matching optimization* is specific to our integrated approach and tightly coupled to the design decision of how to detect and process updates in the data model. Thus, literature does not contain any solutions that are applicable to our approach.

3.2.1.6 Event Detection

The detection of events has been extensively studied for various data storage technologies. For the most common data storage, i.e. relational databases, the *Snoop* system is the best known approach, using an event specification language for active database technology [CM94]. Similarily, Zimmer et. al. support complex update events, which are based on event-condition-action rules [ZMU97]. These approaches support the declarative specification of subscription conditions, but do not propose a high-level specification language based on semantic coherences within the data model. Instead, these specifications are a different means of designing low-level database triggers. Thus, our proposed solution resides on a higher level of abstraction and offers significant advantages to the designers of the event-handling semantics, as we will see when presenting our universal event-handling concept.

A more generic approach, stemming from software technological research, is the type-based approach by Eugster et. al., who use a precompiler that extends any given Java program with publish/subscribe functionality [EGD01]. Amongst other aspects, the detection of events is also part of the research described by Chawathe et. al. [CGL+97, CAW98]: by providing solutions for the detection of events and the subscription to these events, semistructured data bases can be monitored. Again, like for all the solutions we present in this chapter, none of them proposes an integrated concept, as we do in our work.

CQ, a personalized update monitoring toolkit, also presents an approach to event detection by continuously monitoring data sources in the World Wide Web [LPT+98]. The subscription to events in the World Wide Web is also part of Lee's work, which formally describes events and provides triggers to react to those events [LSL00, LSL04]. In contrast to this approach, Cho and Ntoulas investigated how to efficiently detect changes by polling the respective web pages [CN02]. Similarly, Tang showed how to monitor web sources by continously querying them [Tan03]. Due to the nature of web documents (which are monitored by approaches like the above-mentioned), no underlying data model or structure of the respective documents can be taken into account when specifying the subscription semantics. Thus, all the presented publications do not base their semantics on any document structure or data model, which is one of the key points of our approach.

An approach for the monitoring of relational databases has been proposed by Vargas et. al. [VBM05]: based on an XML subscription definition, data updates are monitored and sent to the respective subscribers. A similar approach, using XML as a generic format for publications and XPath for the subscription specification has been presented

by Pereira et. al. [PFJ+01]. Similar to most of the presented related work, these proposals are tightly coupled to technical details of the underlying information system, which is why our approach and the universal event-handling semantics we propose operate on a higher level of abstraction, thus being hardly comparable to the related work and providing an important advantage of generality over existing approaches.

3.2.1.7 Data Storage

Publish/subscribe systems are applied to a variety of different data storages, like relational databases, XML databases, object databases, unstructured information, web pages, etc. We therefore refer to appropriate text books and articles to get insight into these technologies.

3.2.1.8 Subscription and Notification Storage

These aspects are tightly correlated to the subscription specification layer and can be realized in a straight-forward way, following the way notifications and subscriptions are designed. To our knowledge, there are no mentionable publications in this field of interest.

3.2.1.9 Publication Interface

The publication interface, i.e. the way publishers may advertise their events, also depends on the way events are formally described. As a consequence, this aspect of publish/subscribe systems is implicitly dealt with in most publications. Because this aspect is not explicitly in the focus of our work, solutions to provide a publication interface have not been screened separately.

3.2.1.10 Notification Delivery

Considering the delivery of notifications, a number of sophisticated techniques has been developed. Our approach does not consider the aspect of efficiently transporting notifications to the subscribers, so we list several important publications in this field of interest for the sake of completeness.

Since subscribers are usually spatially separate from the event source, events are delivered using network technology. Instead of using static networks, Terpstra et. al. propose a sophisticated peer-to-peer network system named *Rebeca* [TBF+03]. Although this approach focuses on the delivery of notifications, its implicit mechanism of forwarding events along brokers which inspect the event to decide where to forward the notification also covers the field of event matching. *SCRIBE* is another topic-based publish/subscribe system covering the delivery of events (based on the framework *Pastry*) [RKCD01].

Apart from the concept of how to deliver notifications to the respective recipients, the presentation of events can also be an important factor for an event-handling system. Mainly in the field of human interface research, there are several proposals of how to present events to clients: McCrickard et. al. give a nice overview of this area [MCB03, MCSN03], as well as Wahid et. al. [WBL+06].

As soon as heterogeneous information systems, based on different data models, have to be integrated by delivering notifications, particular problems, such as the mapping of the different models onto one common model, have to be solved. *Champagne* provides a prototypic solution to this challenge [RCHM02].

3.2.1.11 Security

Finally, several publications deal with the security of publish/subscribe systems. One approach therefore puts a network of trust over the participating brokers of a distributed architecture, where public-/private-key pairs are used to encrypt the events sent between the different network nodes [FZB+04], to name only one.

3.2.2 Commercial Systems

A proposal that fits well into our scenario has been patented by Oracle Corporation [Deu04] for the Oracle Database: by providing the possibility to specify boolean filters based on the SQL specification of any database schema, users are enabled to specify their interest and can automatically be notified about relevant updates. However, this approach does not fulfil all of our requirements, especially when it comes to the semantic concepts like implicit subscriptions or event-propagating references.

A commercial approach using an architecture similar to our proposal is integrated into Microsoft SQL Server from version 2005 on, called *Microsoft SQL Server Noti-*

fication Services [Micb]. Based on a specification of subscriptions using SQL, events can be detected and sent to subscribers along predefined channels. Although this overall approach fits our technical use case descriptions, functional and non-functional requirements like the ease-of-use or the possibility to describe subscription semantics on a high level (i.e. as an enhancement of a model of the database tables) are not fulfilled.

A solution developed by the third major database vendor, *IBM*, offers a middleware framework for the detection and handling of events, called *Amit* [ABEYH00]. Together with the respective authoring tool, so-called "situations" can be defined which are then used to determine relevant subscribers and inform them about events that match these situations. Although being an interesting solution with the maturity of an established product, the underlying concepts do not provide universally applicable concepts for the specification of event-handling semantics. In particular, this approach lacks the abstraction level which is necessary to realize a general, vendor- and technology-independent event-handling framework, as our approach does.

3.2.3 Comparison Against Requirements

Table 3.1 finally shows the results of our evaluation in a compressed form. Only publications from the above list fulfilling several of our requirements have been considered in this table. [2]

[2] Although our requirements /R1.1/ and /R1.2/ talk about entities and/or classes in general, the presented solutions deal with different kinds of entities, such as web pages etc. Therefore, we interpret the term "entity" individually, as it is appropriate.

	/R1.1/	/R1.2/	/R1.3/	/R1.4/	/R1.5/	/R1.6/	/R1.7/	/R1.8/	/R2.1/	/R2.2/	/R2.3/	/R2.4/	/R2.5/	/R2.6/	/R3.1/	/R3.2/	/R3.3/	/R3.4/
Le Subscribe [PFLS00, PFL+00]	✓		✓		✓										✓			
CDWeb [FFM01]	✓		✓		✓				✓	✓		✓			✓			✓
"User Notification ..." [BFRS06]	✓	✓			✓	✓						✓	✓		✓			
EQAL [EDS+04]	✓	✓			✓									✓			✓	
EQAL [Wan08]											✓				✓	✓	✓	
OPS [WJL04]	✓		✓		✓						✓	✓						
Cream [CBB03]			✓		✓													
"Content-Based ..." [EG01]	✓		✓		✓						✓					✓	✓	
"On Objects and Events" [EGD01]	✓		✓		✓											✓		
"Event and rule services ..." [LSL04]	✓		✓		✓				✓	✓	✓				✓			
"Integrating Databases ..." [VBM05]	✓		✓		✓				✓	✓					✓			
WebFilter [PFJ+01]	✓		✓		✓				✓	✓	✓	✓			✓			
Scribe [RKCD01]					✓										✓			
Oracle [Deu04]	✓		✓		✓				✓									
Microsoft SQL Server [Micb]	✓		✓		✓				✓	✓	✓				✓			✓

Table 3.1: Comparison of related work against our requirements

As we can see, none of the evaluated approaches and systems completely fulfils our requirements. Obviously, this is no urgent reason to completely develop a new approach, but instead re-use ideas and concepts from previous researches, which is what we will do on a conceptual level. However, none of the above-mentioned approaches focuses on information systems where publishers and subscribers "reside" within the same data storage and use the same data model, so that reuse will be limited to a few aspects.

In particular, none of the available solutions offers an integrated solution framework considering the declarative specification of subscriptions based on an information system's data model and the automatic generation of the respective optimized update event detectors, which is what has been developed during our research and makes up the novelty of our approach.

3.3 Summary

Based on the common scientific classification of publish/subscribe systems in combination with our particular use cases, we developed a detailed architectural model. This model was used as a basis for the classification of several scientific and commercial approaches. Those approaches were also compared against the requirements we developed in the previous chapter. This comparison clearly yielded the need for the development of a new solution, which will be presented in the following parts of this dissertation.

Part II

The Non-Invasive Approach

"If the only tool you have is a hammer, you tend to see every problem as a nail."

Abraham Maslow

4

Solution Overview and Generic Architecture

In the following chapter, we give a bird's eye view onto our solution. After motivating why we chose to develop a generic, generative and declarative approach, we present our solution: we show how a declarative specification can be transformed into an event-handling runtime component that centrally monitors the data storage of an information system and determines all relevant subscribers for detected updates. The different layers of abstraction that are inherent to our solution will also be highlighted. Next, the different architectural parts of the event-handling component are presented in detail. Finally, we clarify the dynamic aspects of our solution's lifecycle, looking at both design time processes and runtime processes. This chapter concludes with a brief summary, collecting the key features of our solution.

4.1 Motivation for the Generative Approach

Before presenting our approach, we are briefly going to explain why we chose a generative approach in conjunction with a suitable meta-model. Although this discussion can be lead independent of any particular implementation strategy, we are going to

illustrate our considerations by explaining the different aspects in the course of the development of an event-handling system for active database systems, i.e. building a notification system using database triggers.

The first argument for a generative approach is the observation that event handling constitutes a typical cross cutting concern [TOHSMS99, BLS03]: since data updates can stem from a multitude of spots in the application code, in a conventional approach each of those spots would have to be monitored by writing the additional monitoring and notification code fragments at the appropriate locations. Figure 4.1 visualizes this multitude of modifications in a traditional three-tier architecture, affecting the data access layer as well as the application layer itself.

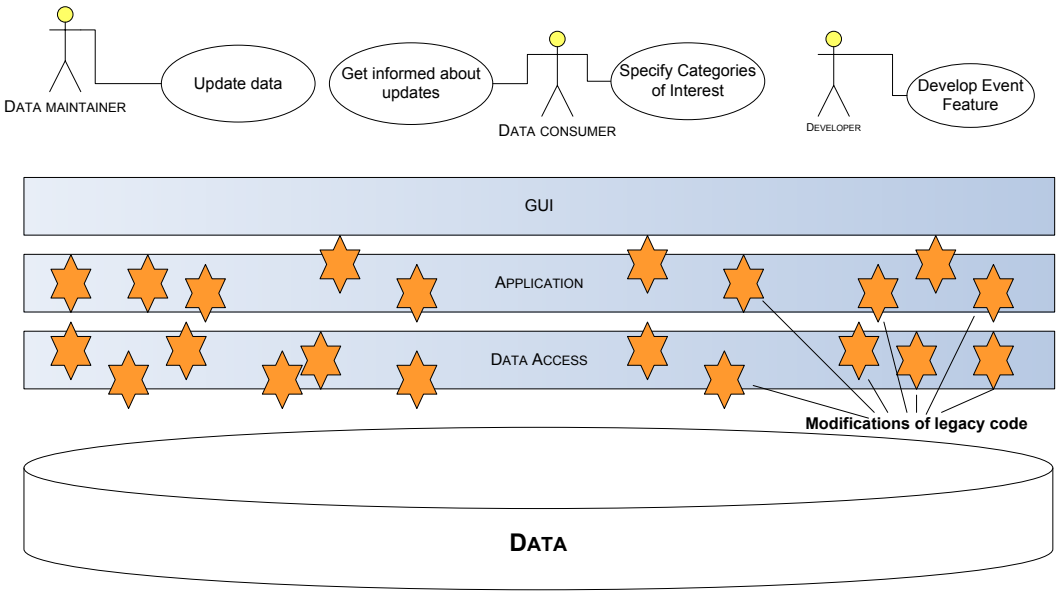


Figure 4.1: Event-handling as a cross cutting concern

Various techniques to handle cross cutting concerns exist, for instance Aspect Oriented Programming (AOP) [Ecl]. Using AOP, the monitoring code can be automatically "woven" into those fragments of the application where data modifications are performed. However, we could show in [GF05] that AOP may solve the problem of maintainability, but a problem of bad performance (because of the high-level of implementation) still remains: if source code is modified to detect updates, the monitoring code is located at a very high level of abstraction, so that the application is notice-

ably slowed down. In addition, even AOP requires access to and knowledge about the application's code, which can not always be assumed as given.

Instead, a central observation of the data store proved to be more suitable. But this strategy causes an other problem: the multitude of different triggers (in case relational databases are used), or similar monitoring fragments which have to be developed, must be handled. Not only arises the need to develop such a trigger for every individual observed entity, but - as soon as standard software has to be developed - also for every target platform, i.e. for any database vendor, as illustrated in figure 4.2. Obviously, it is almost impossible to develop and maintain all these triggers with adequate effort, at least as soon as modifications of the application and/or the event semantics have to be handled.

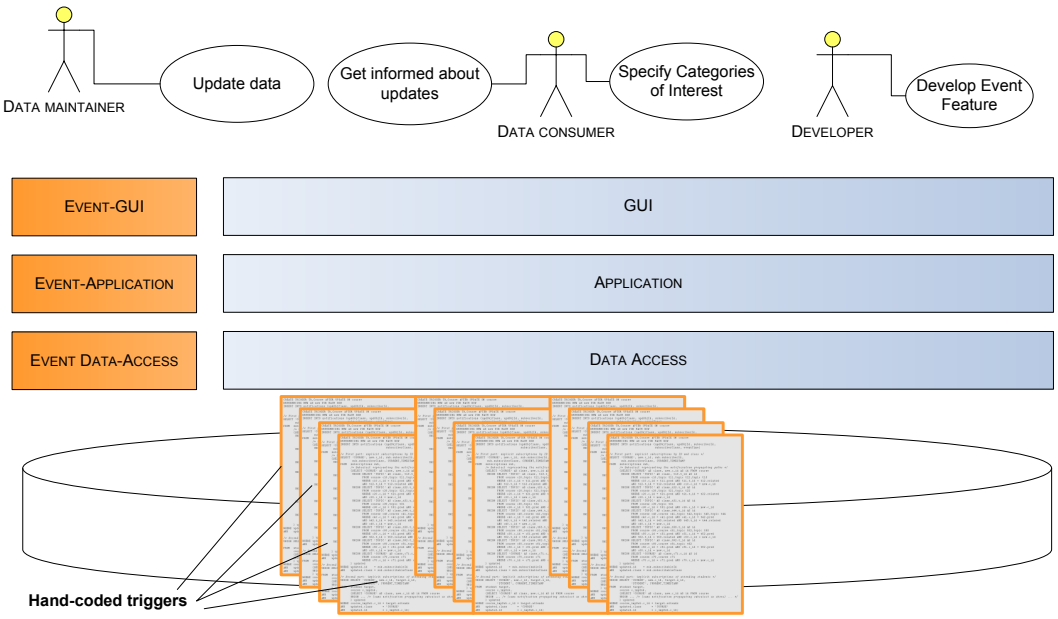


Figure 4.2: Necessity for a multitude of database triggers

In addition, specialized knowledge in the field of trigger development is required to write performant and maintainable trigger code - for every individual database system. This again raises the need to use a generative approach so that this particular knowledge can be centrally realized in the form of templates and transformations, generating "good" code automatically and by design.

If we consider a third aspect, the fact that event-handling semantics are usually strongly correlated to the application's underlying data model (as we found out during the use case analysis), a generative approach that uses the data model as a basis and automatically generates the monitoring triggers offers additional advantages.

Thus, we decided to develop a declarative and generative approach, which is introduced in the following section.

4.2 The Declarative-Generative Approach

Our approach provides the possibility to generate event-handling code from a declarative specification of the respective application semantics. This specification is based on a precise formal model, specifying how updates should be monitored and interested subscribers have to be found. Since we found out that the event-handling semantics are strongly correlated to the underlying data model, we base our event-handling model on a generic meta-model, as we will show in chapter 5.

As a counterpart to the meta-model, we also provide the ability to automatically transform the abstract specification into the corresponding fragments of event-handling code.

Before we explain the lifecycle steps, the four participating roles are briefly sketched:

- *Framework Developers* provide the meta-model mentioned above, as well as the respective transformations. We will present the abstract model for such a meta-model, i.e., the meta-meta-model, and provide an exemplary implementation of the meta-model, based on UML, as well as a prototypic implementation of the model-to-code-transformations for active database systems.
- *Developers* use a suitable meta-model (i.e. an instance of the meta-meta-model) and the transformations to enrich the data model of any arbitrary information system and automatically generate the respective event-handling code.
- *Data Consumers* are users of the underlying information system, using the provided event-handling functionality in addition to the regular functionality of the system.
- Finally, *Data Maintainers* are also users of the information system, who modify data using the application and thus may trigger event-handling functions, if intended by the developers.

Figure 4.3 illustrates our overall approach and the corresponding development lifecycle, which will be explained in detail in the following.

4.2.1 Central Data Storage

According to our use case analysis, we assume that there is a central data storage containing all data that have to be monitored (1). As we will see later, our solution is completely independent of the type of storage, be it relational databases, XML databases, flat files, or anything similar.

4.2.2 Information System's Data Model as a Basis

The only prerequisite of our approach is that the storage contains (semi-)structured data, i.e. that some model of the stored data exists or can at least be derived from the information system. However, our approach is not limited to a particular formalism for this model, i.e. ER-diagrams, UML models, XSchemata or any other type of formal model that suits the data storage can be used. In figure 4.3, we use a small sample UML model for illustration purposes (2).

4.2.3 Generic Meta-Model of Arbitrary Data Models

As long as the information system contains data in a structured form, so that a model of the stored data exists, we can further assume that there is a suitable meta-model. We do not rely on a particular kind of meta-model; as an example, figure 4.3 depicts a UML meta-model illustration (3).

What we do demand, however, is the existence of concepts resembling *Entities*, *Attributes* and *Associations* within the chosen meta-model. Since all known meta-models support this requirement, we do not consider this as a significant restriction to the universality of our approach.

4.2.4 Enhancement of the Meta-Model by a Generic Event-Handling Meta-Model

This step is where the *framework developers* come into play: The meta-model of the information system has to be enhanced by a formalism for the additional constructs

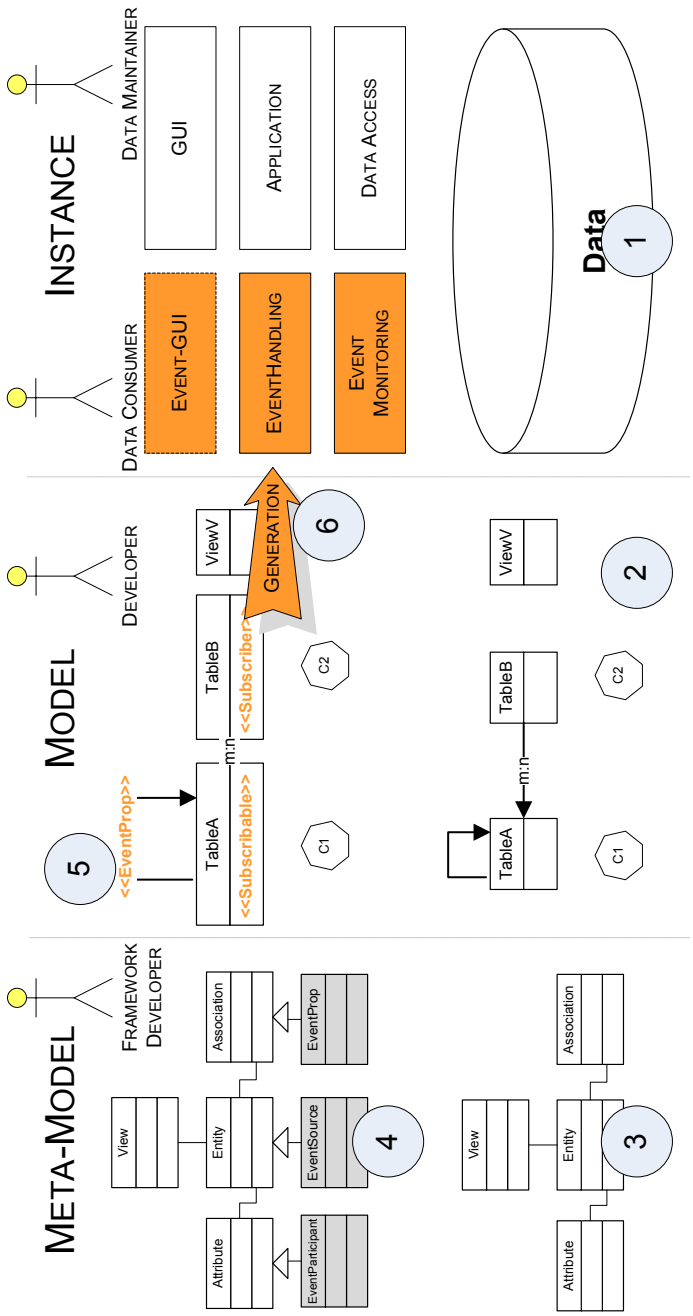


Figure 4.3: Illustration of the declarative-generative approach

of our event-handling semantics, which will be presented in chapter 5 (4). It depends on the type of meta-model how this can be done; an exemplary enhancement for UML profiles will be presented in part III of this dissertation.

Corresponding to the enhanced meta-model, a set of transformation routines have to be developed by *framework developers*. These transformations will later be used to take the enhanced meta-model as an input and generate the event-handling code fragements which actually realize the additional notification functionality in the information system under development. In this dissertation, we will present a suitable prototypic implementation for active database systems in part III.

4.2.5 Enrichment of the Data Model

Developers can then use this enhanced meta-model to enrich the meta-model of the respective information system with the adequate event-handling semantics, based on the "vocabulary" which has been developed by *framework developers*. This additional tagging of the meta-model creates the so-called event-handling *overlays* which constitute the basis for the next step of our approach.

In our illustration (fig. 4.3), this enrichment is visualized by UML stereotypes, tagging entities, attributes and associations (5).

4.2.6 Transformation of the Enriched Data Model into Event-Handling Code

In a final step, the transformations provided by *framework developers*, which have to be tailored to the type of data storage which is in use, can then be applied to the tagged data model (6). The following additional components of the information system, which are responsible for the notification facility, can be generated from the descriptive semantic specification without the need for any hand-written code:

- The *event monitoring* component, responsible for the detection of updates in the data storage,
- the *event-handling* component, realizing the matching of events and the determination of the respective subscribers and
- the *event GUI*, which serves as a user interface for *data consumers* and *data maintainers*. This GUI is mainly responsible for the specification of subscriptions

and the presentation of notifications, however, this component is not in the focus of our work.

4.3 Genericity and Dimensions of Abstraction

Up to now, we only claimed the genericity of our solution. In the following, we will discuss why our approach is truly generic by presenting the different dimensions of abstraction leading to this genericity. Figure 4.4 illustrates the two abstraction dimensions[1].

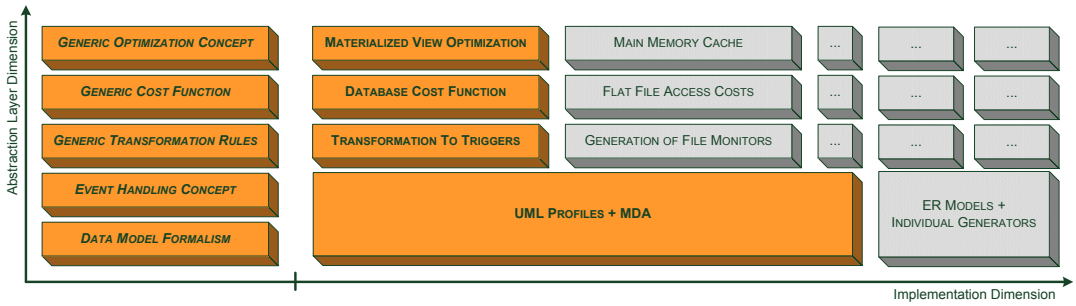


Figure 4.4: Dimensions of abstraction

We observe two dimensions of abstraction: the *abstraction layer* dimension and the *implementation* dimension.

The first dimension represents the traditional software layer abstractions. From the bottom up, we first encounter the data model of the information system, which constitutes the basis for the event-handling system. On top of the data model resides our generic event-handling concept, which will be presented in the next chapter. The different constructs of this concept then have to be transformed into the event-handling code, which will be described on an abstract level in chapter 6 of this dissertation. The top two layers contain a generic cost model for the event matching procedure, as well as a generic way of how to optimize the matching. Both will be presented in chapter 7. In brief, the leftmost column of figure 4.4 thus represents the generic and implementation-independent part of our work.

[1]Coloured building blocks are described in this dissertation, whilst grey entities are possible extensions and implementation alternatives which will not bepresented in the following.

The second dimension of abstraction concerns the different ways in which these abstract concepts can be implemented, visualized by the x-axis in figure 4.4. Each of the abstraction layer elements can be realized in a variety of ways, wherein implementation alternatives can - under certain circumstances - be recombined and re-used.

In our approach, which we will present in part III, we will use UML profiles in conjunction with Model Driven Architecture to specify the data model and the event-handling semantics. However, different formalisms, for instance entity relationship models, as well as different ways to extend these models and generate code from them, are possible, too. The generic transformation rules can also be applied to a variety of target systems: besides the generation of relational database triggers, monitoring threads for flat files, XML database monitors, and many more are imaginable. The same variety of possibilities exists for the concrete cost model (in our case, we will use a model based on database access plans) and for the way in which optimizations are realized - materialized views are one possible solution which we will examine in detail.

These two layers of abstraction also enable different dimensions of extension to our concept: while the five-tier architecture can be extended to add different functionality (or modified at any layer), different implementations of all these concepts are realizable, thus making the overall approach very powerful, extensible and adaptable to any new requirements.

4.4 Details on the Architecture's Components

Figure 4.4 implicitly reveals the basic building blocks of our event-handling system's architecture. In the following, we will zoom into these aspects and present the generic refined software architecture of our approach. We will also show where the different kinds of models are derived from, where they are used and which components undertake the tasks we presented in the classification scheme in chapter 3.

Figure 4.5 presents the overall architecture, distinguishing between designtime components (upper section) and runtime components (lower section).

4.4.1 Models

We can identify three different models: on the one side, the data model (which can be extracted from the actual data storage or be specified by the developer) as well as the event model (which is specified by the developer) represent the semantics of the

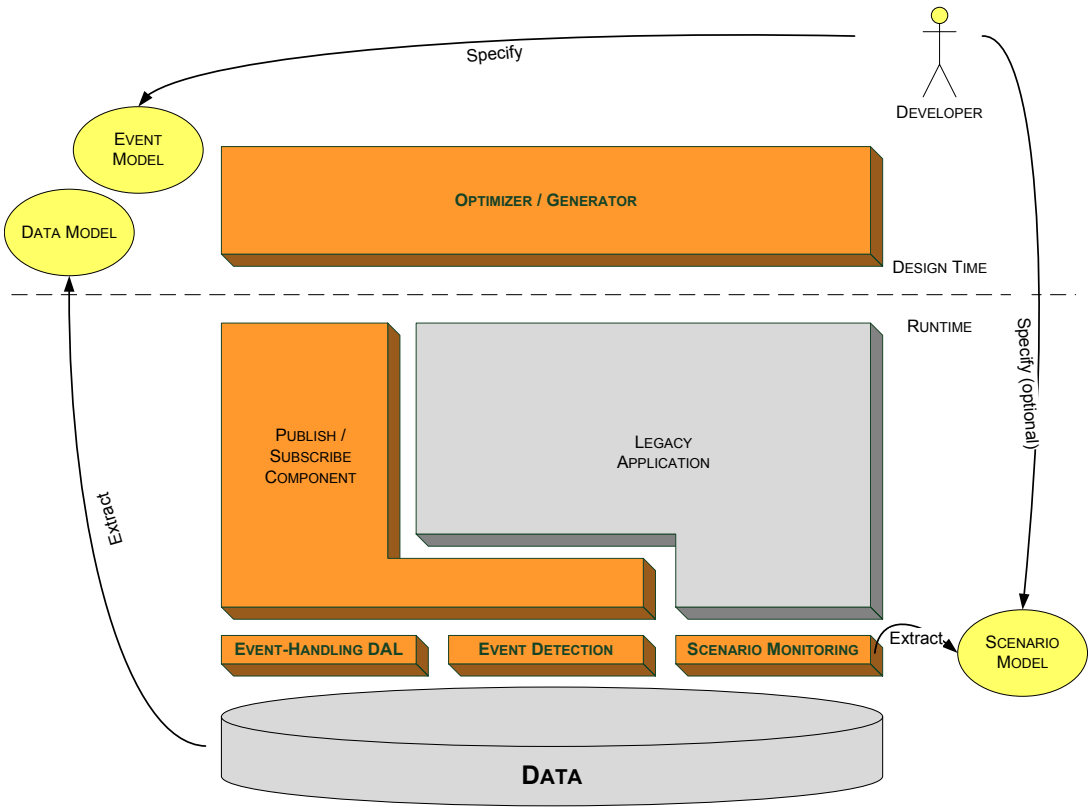


Figure 4.5: System architecture

event-handling component. These two models constitute the input of the generator, which creates all relevant components (coloured boxes in the runtime section of the architecture graphic). On the other side, a scenario model contains information about the actual usage statistics of the legacy application, which will serve as an input to the optimizer and can either be automatically created by the scenario monitor or manually be specified by the developer, as we will show in chapter 7.

4.4.2 Event-Handling Data Access Layer

The publish/subscribe component must be able to access legacy data (in order to accept subscriptions and present notifications, as we will see later on), so that a suitable data access layer must be available. Since information about the data model is known, it is easily possible to generate this data access layer from the model specification

automatically: any information needed for this generation step is present in the (extracted or manually specified) data model. Depending on the type of data storage, the respective layer implementation can easily be automatically created. Recalling our classification scheme, this layer thus implements *Subscription Storage* and *Notification Storage*.

4.4.3 Event Detection

Located between the legacy application, the publish/subscribe component and the data storage, this component is responsible for the detection of relevant update events. It is also automatically generated by the optimizer/generator and thus implements the semantic specification for event-handling in an optimized way. This component therefore realizes the aspects *Event Detection*, *Event Matching* and *Matching Optimization* of our classification scheme.

4.4.4 Scenario Monitoring

The scenario monitoring is located between the legacy application and its data storage. It can be implemented in two ways: either as an additional layer between the legacy application's data access layer and the application (as a kind of "virtual" data access facading the actual data access layer), or as a stand-alone component using data storage hooks (e.g. triggers) to monitor access patterns. Its results are then stored in the scenario model, containing information about read- and update frequencies. This component plays a special role in the overall architecture, since there is no equivalent in the classification scheme. However, it provides the necessary data for optimization steps and can thus be also subsumed under *Matching Optimization*.

4.4.5 Publish/Subscribe Component

The publish/subscribe component is realized as a stand-alone component giving users access to the event-handling functionality. This component's main purpose is to allow users to specify their subscriptions and to deliver the notifications to them, so that we can match this component to the classification areas *Subscription Interface* and *Notification Delivery*.

The user interfaces for subscription and notification delivery are mainly a particular way to display entities of the application's data model: subscriptions can be seen

as the presentation of arbitrary entities, together with a flag "subscribe to", while notifications are nothing more than a presentation of the updated entity together with status information (who modified the entity, when did he do so, what was the modification). Thus, these GUIs can also be automatically generated by the generator, taking the data and event model as an input, but this is not in the focus of our research.

4.4.6 Optimizer/Generator

At design time, the combined optimizer/generator represents the core of our approach: using the three models as an input, every component of the event-handling system can be generated. Both the generator as well as the implicit optimization techniques will be described in detail in the remainder of this dissertation, so we will not go into detail here.

4.4.7 Legacy Application

Finally, the legacy application constitutes the information system that has to be enhanced with event-handling functionality. For our approach, it does not matter whether it is an in-force system which is already used, or whether it is a software system that is still under development.

4.5 Dynamic View on the System's Lifecycle

To complete our overview, we will leave the static aspects and take a look at the dynamic properties of our concept. To do so, we will briefly describe the designtime and runtime lifecycle in order to give insight into its dynamic behaviour.

4.5.1 Designtime Lifecycle

Figure 4.6 visualizes the different steps and the input and output data that are relevant during the design time lifecycle.

In case a (legacy) application already exists, the corresponding data model has to be extracted. If there is no such application, the data model has to be specified manually

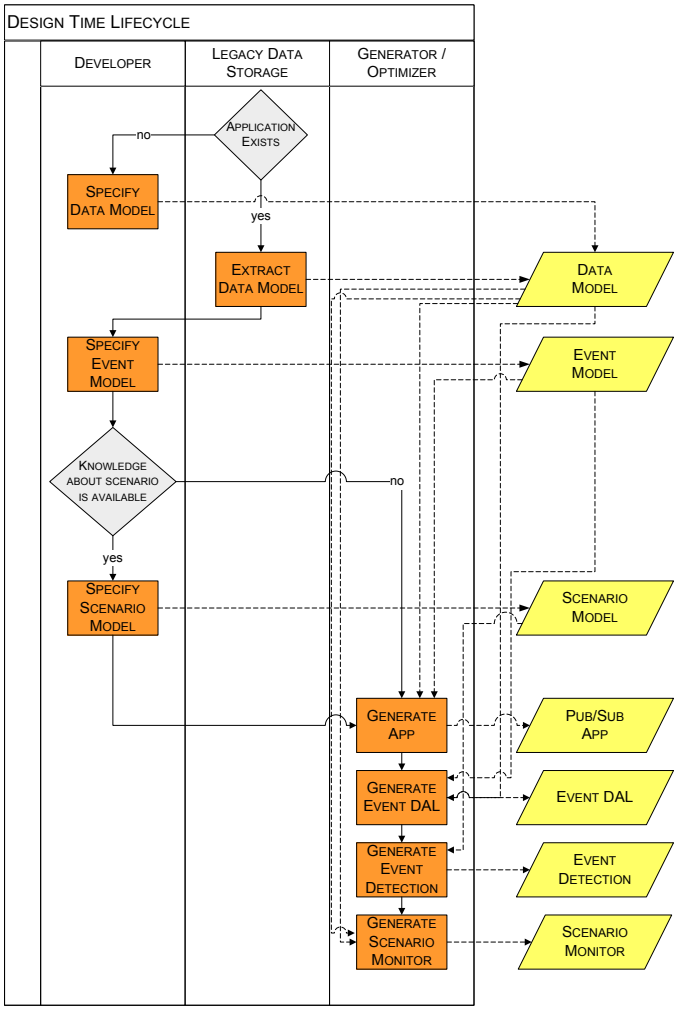


Figure 4.6: Designtime lifecycle

by the developer. As a next step, the semantic event model has to be created by a developer.

As we will see in chapter 7, information about the data access patterns, stored in the scenario model, can be used to generate an *efficient* event matching component. If the developer already has knowledge about these patterns at designtime, he can store

this information in the respective scenario model. If no such model can be built, the event-handling component will be created without any in-advance optimizations.

Finally, all architectural components are generated: the event-handling application itself, based on the data model and the event model, the event data access layer using information about the data model and the event model, the event detection component which uses all three models, and finally the scenario monitor, using the data and event model.

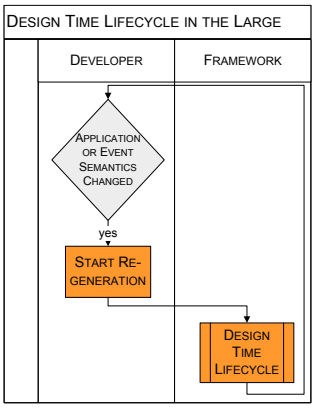


Figure 4.7: Designtime lifecycle in the large

Since both the application as well as the event-handling semantics tend to change over time, the subsequent applicability of our approach also finds its way into the designtime lifecycle in the large, as visualized in figure 4.7: whenever the information system or the desired event-handling semantics change, the re-generation of all artifacts can be started by developers without any additional effort.

4.5.2 Runtime Lifecycle

Finally, the runtime lifecycle, presented in figure 4.8, has to be examined.

Even if the application and/or event model should not change over time, so do the data access patterns. To find that out, the access patterns are constantly monitored by the scenario monitor. As soon as a significant change is detected, an administrator can then start the re-generation, so that the new insights can be used to generate a better suited implementation of the event detection component. As a further improvement, it would also be possible to programmatically detect significant changes by comparing

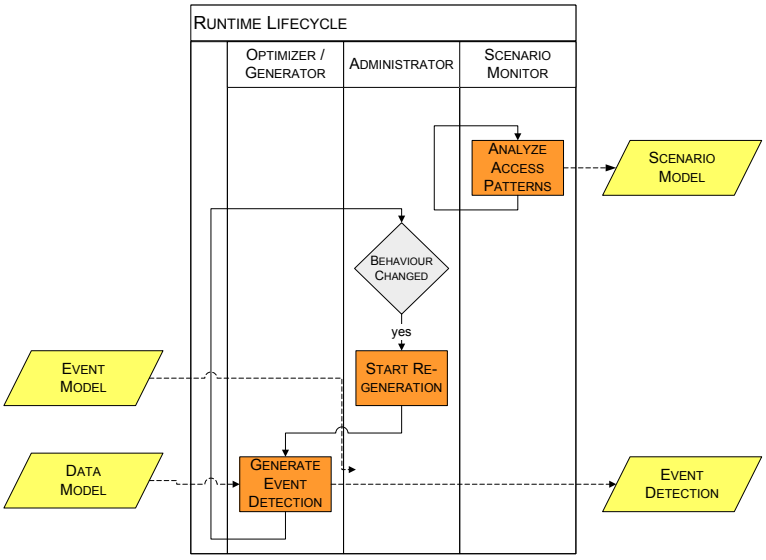


Figure 4.8: Runtime lifecycle

the scenario model over time and automatically start the regeneration process, but this is beyond the focus of our work.

4.6 Summary

In this section, we presented our generative approach from a high-level view and showed its genericity and the different dimensions of abstraction. The architecture was presented both from a static and a dynamic viewpoint to demonstrate its overall behaviour.

In the following, we will go into more detail and present our meta-meta-model, the algorithms to generate the monitoring functionality and a generic optimization approach.

"If you can't explain it in five minutes, either you don't understand it or it doesn't work."

Darcy McGinn

5

Conceptualization of Data and Event Models

The following section presents the conceptualization of our approach. We will introduce notations for information system models and for the representation of instances of information systems, i.e. for entities and their attributes stored within the system. Based on these representations, we will present the semantic concepts of our event-handling approach, called *implicit subscriptions*, *explicit subscriptions* and *event-propagating associations*. Furthermore, we will define how these event-handling constructs have to be interpreted to handle data updates and determine the respective subscribers. To illustrate the concepts, an example will be presented, along which we will explain how to interpret the semantic concepts for several sample updates.

5.1 Representation of Data Model and System Instance

5.1.1 Data Model

In the object oriented world, the model of a software system is basically represented by *classes*, their *attributes* and *associations* between classes. Classes are denoted by capitalized names, such as Lectures and Documents.

Attributes of classes are typed and represented using the dot-notation:

[classname].[attributename]:[type]

If class Lectures contains an attribute title of type String, this would be denoted by

Lectures.title:String.

Associations between classes are represented similarly to attributes, with the association target class being the type of the attribute. If a $1 : n$ or $m : n$ association is represented, the type is usually represented by a collection type. The association belongsTo between a document and *zero or exactly one* lecture is thus represented by

Documents.belongsTo:Lectures,

whereas the association attends between students and *many* lectures is denoted by

Students.attends:Collection<Lectures>.

Associations between classes are always binary and directed, i.e. the class containing the attribute is the source class, while the attribute type (or the collection type) is the target of the association. As a consequence, bi-directional associations have to be represented by one attribute per class, where each class is once the source and once the target of the association.

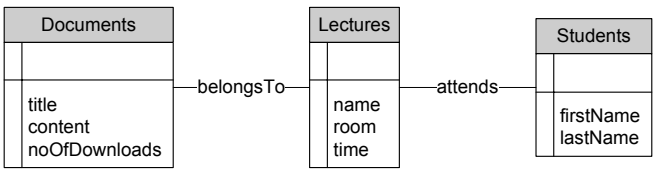


Figure 5.1: Sample data model

Figure 5.1 shows a sample data model, where both associations belongsTo and attends are bi-directional. The corresponding formalization of this model is:

Documents.title:String
Documents.content:URL
Documents.noOfDownloads:Integer
Documents.belongsTo:Collection<Lectures>

Lectures.name:String
Lectures.room:String
Lectures.time:Time
Lectures.belongsTo:Collection<Documents>
Lectures.attends:Collection<Students>

Students.firstName:String
Students.lastName:String
Students.attends:Collection<Lectures>

5.1.2 System Instance

Instances of a data model are represented by objects and their attribute values, which implicitly also contain the references between objects. An instance of an arbitrary class is denoted by

[ClassName]:[objectId].

For example, the objects doc1 of class Documents and lectureA of class Lectures are represented by

Documents:doc1

Lectures:lectureA

Attribute values are described by

[ClassName]:[objectId].[attributeName] = [attributeValue]

To describe the contents of collections, we use a set-based notation:

[ClassName]:[objectId].[attributeName] = {[value1],[value2],...,[valueN]}

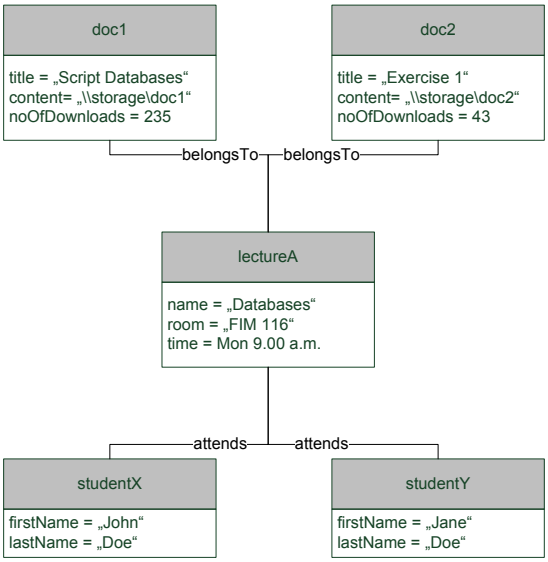


Figure 5.2: Sample system instance

The sample instance presented in figure 5.2, corresponding to the above-mentioned data model, is thus represented by:

Documents:doc1.title = 'Script Databases'
Documents:doc1.content = '\\storage\doc1'
Documents:doc1.noOfDownloads = 235
Documents:doc1.belongsTo = {Lectures:lectureA}

Documents:doc2.title = 'Exercise 1'
Documents:doc2.content = '\\storage\doc2'
Documents:doc2.noOfDownloads = 43
Documents:doc2.belongsTo = {Lectures:lectureA}

Lectures:lectureA.name = 'Databases'
Lectures:lectureA.room = 'FIM 116'
Lectures:lectureA.time = Mon 9.00 a.m.
Lectures:lectureA.belongsTo = {Documents:doc1, Documents:doc2}
Lectures:lectureA.attends = {Students:studentX, Students:studentY}

Students:studentX.firstName = 'John'
Students:studentX.lastName = 'Doe'
Students:studentX.attends = {Lectures:lectureA}

Students:studentY.firstName = 'Jane'
Students:studentY.lastName = 'Doe'
Students:studentY.attends = {Lectures:lectureA}

5.2 Event-Handling Constructs and Formal Event Model

Starting from this formalism, we will next introduce our event-handling concepts, which are presented in the following section. For a brief illustration of the event-handling constructs, we use the above-mentioned data model and the following semantic requirement, visualized in figure 5.3:

"Whenever the content of a document is updated, any lecture referring to it has to be considered as updated, too. Additionally, all attendees of a lecture should automatically be informed about such an update."

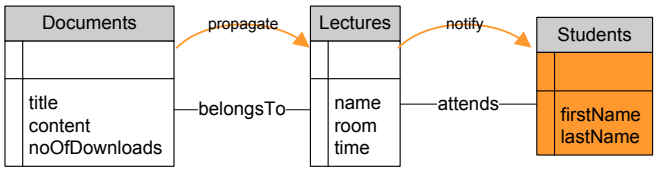


Figure 5.3: Sample event semantics

In our sample instance, this means that any update of the contents of doc1 or doc2 would automatically cause studentX and studentY (cf. figure 5.4) to be informed.

This scenario contains the elements described in the following:

5.2.1 Subscribers

All classes whose objects are meant to be able to receive events are called *Subscribers*. By this means, requirement /R1.1/ is respected.

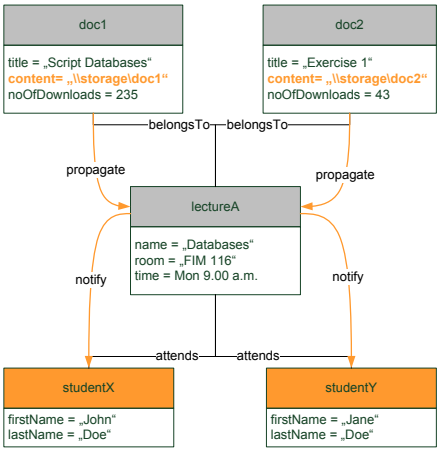


Figure 5.4: Impact of event semantics on the system instance

Definition 5.2.1 (Subscribers) Let [className] be a class in the object model that should be a possible receiver of update events. We call [className] a *subscriber* and denote this by

≪Subscriber≫[className].

If a class is marked as a subscriber, this means that every instance of this class is a subscriber, i.e. we use the higher abstraction level of the data model to generically handle all possible instances of this data model.

In our scenario, we write

≪Subscriber≫Students.

5.2.2 Subscribables

As a dual concept, *Subscribables* represent any class that can be the source of a handled update event that has to be monitored, as requirement /R1.2/ demands.

Definition 5.2.2 (Subscribables) Let [className] be a class in the object model that has to be monitored for updates. We call [className] a *subscribable* and represent this by

≪Subscribable≫[className].

Again, the class in the data model is tagged, meaning that all instances of this class have to be considered as subscribables.

5.2.3 Observed Attributes

Due to requirement /R1.3/, only updates of individual attributes should be handled, so we use a similar notation for such attributes:

Definition 5.2.3 (Observed Attributes) Let ≪Subscribable≫[className] be a subscribable class in the data model, as introduced in definition 5.2.2, let [observedAttribute] be an attribute of this class that should be checked for modifications. We call [observedAttribute] an *observed attribute* of class [className] and denote this by

≪Subscribable≫[className].≪Subscribable≫[observedAttribute].

In the above case, we write

≪Subscribable≫Documents.≪Subscribable≫content,

meaning that the attribute content of every instance of Documents has to be monitored.

5.2.4 Implicit Subscriptions

To express that instances of a particular subscriber class should implicitly be informed about updated objects of an associated subscribable class (requirement /R1.4/), we introduce *implicit subscriptions*.

Definition 5.2.4 (Implicit Subscriptions) Let [associationName] be an association in the data model from a subscribable class ≪Subscribable≫[sourceClass] to a subscriber ≪Subscriber≫[targetClass]. If updates of an instance of [sourceClass] should implicitly lead to notifications of all instances of [targetClass] that are referencing the updated object via [associationName], we call [associationName] an *implicit subscription*, denoted by

≪Subscribable≫[sourceClass].≪implicitSub≫[associationName]:≪Subscriber≫[targetClass]

As with previous concepts defined on the data model, all instances of [targetClass] that are linked to an updated instance of [sourceClass] via the association [associationName] have to be notified implicitly.

Due to this definition, we demand that only existing associations from *subscribables* to *subscribers* can be declared to be implicit subscriptions.

The implicit subscription attends between Lectures and Students from figure 5.3 is thus expressed by

≪Subscribable≫Lectures.≪implicitSub≫attends:≪Subscriber≫Students.

5.2.5 Event-Propagating Associations

If an update of a subscribable object should automatically cause all subscribable objects that are associated via a particular association (requirement /R1.6/), we introduce the concept of *event-propagating associations*:

Definition 5.2.5 (Event-Propagating Associations) Let [associationName] be an association in the data model from a subscribable class ≪Subscribable≫[sourceClass] to another subscribable class ≪Subscribable≫[targetClass]. If updates of an instance of [sourceClass] should automatically lead to an update of all instances of [targetClass] that are linked to the updated object via the association [associationName], we call [associationName] an event-propagating association, denoted by

≪Subscribable≫[sourceClass].≪eventProp≫[associationName]:≪Subscribable≫[targetClass].

This definition implies that only existing associations between two *subscribables* can be tagged like this.

Further on, the impact of an event propagation has to be limited, as identified in requirement /R1.7/. We therefore introduce the integer value attribute impactRange for event-propagating associations.

Definition 5.2.6 (Impact Range) Let ≪eventProp≫[associationName] be an event-propagating association between the subscribable classes ≪Subscribable≫[sourceClass] and ≪Subscribable≫[targetClass]. If the update event should be propagated from instances of [sourceClass] to instances of [targetClass] along at most i references of type [associationName] (the detailed semantics will be explained in section 5.5), this is expressed using the *impact range* of an event-propagating association, denoted by

≪Subscribable≫[sourceClass].≪eventProp @impactRange=i≫[associationName]:

≪Subscribable≫[targetClass].

Representing figure 5.3, we write

≪Subscribable≫Documents.≪eventProp @impactRange=1≫belongsTo:≪Subscribable≫Lectures

5.2.6 Explicit Subscriptions

In contrast to the above constructs, explicit subscriptions between *subscribables* and *subscribers* (requirement /R1.5/) are not expressed in the data model, but as tuples of objects, i.e. they are part of the system instance.

Definition 5.2.7 (Explicit Subscriptions) Let

[sourceObject]:≪Subscribable≫[sourceClass]

be a subscribable object,

[targetObject]:≪Subscriber≫[targetClass]

be a subscriber object. If an update of the instance [sourceObject] should lead to a notification of the subscriber instance [targetObject], this is represented by a tuple

(source:≪Subscribable≫[sourceClass], target:≪Subscriber≫[targetClass]),

called an *explicit subscription*.

The set of all explicit subscription in a system instance is denoted by

$$Sub_{explicit} = \{exp_1, ..., exp_n\},$$

with exp_i defined as above.

A set containing the single explicit subscription between document doc1 and student studentX would thus be expressed by

$$Sub_{explicit} = \{(\text{doc1:≪Subscribable≫Documents, studentX:≪Subscriber≫Students})\}.$$

5.3 Overlays

The usage of subscribers, subscribables, implicit and explicit subscriptions and event-propagating associations can only represent one particular aspect of an event-handling

system at a time. Since real-life use cases may contain several different semantic aspects, *overlays* are introduced.

An overlay has two functions:

- it describes *which subscribable* has to be monitored
- it describes *how* detected updates of this subscribable have to be handled, considering implicit subscriptions and event-propagating associations

An overlay can be represented as a 3-tuple, containing a subscribable class, a set of implicit subscriptions and a set of event-propagating associations.

Definition 5.3.1 (Overlays) An overlay $\mathcal{O}_i$ is defined as a 3-tuple

$$\mathcal{O}_i = (\text{[subscribableClass]}, A_{implicitSub}, A_{eventProp})$$

where [subscribableClass] is the subscribable class that has to be monitored for updates,

$$A_{implicitSub} = \{imp_1, ..., imp_i\}$$

denotes the set of implicit subscriptions that have to be considered within this overlay and

$$A_{eventProp} = \{prop_1, ..., prop_j\}$$

represents all event-propagating associations that have to be handled in $\mathcal{O}_i$.

The overlay from figure 5.3 monitoring documents, propagating updates from documents to lectures and implicitly notifying all attendants of a lecture is thus expressed by

$$\mathcal{O}_1 = (\text{≪Subscribable≫Documents},$$

$$\{\text{≪Subscribable≫Lectures.≪implicitSub≫attends:≪Subscriber≫Students}\},$$

$$\{\text{≪Subscribable≫Documents.≪eventProp @impactRange=1≫belongsTo:≪Subscribable≫Lectures}\}).$$

The overall semantics of an event-handling system can be expressed by a set of overlays

$$\mathcal{O} = \{\mathcal{O}_1, ..., \mathcal{O}_n\},$$

with one overlay representing exactly one aspect of the event-handling intention.

5.4 Graph Representations for Data Model, Overlay and Instance

The overall semantics of the above-mentioned constructs can be explained best using graph-based representations of the data model, the event-handling overlays and an arbitrary system instance, which will be introduced in the following.

5.4.1 Graph Representation of Data Model

The graph representation of a data model is expressed as the directed *model graph* $G = (C, A)$ consisting of a set of class vertices $C = \{c_1, ..., c_n\}$ and association edges $A = \{a_1, ..., a_m\}$. To create the model graph for a given data model, a vertex c_i labeled with the class name and its tags, if any, has to be added to the graph for every class within the data model. Similarly, for every association in the data model, a directed edge a_j between the corrsponding source class node and the target class node has to be added. Associations are labeled with the name of the association.

Figure 5.5 shows the graph representation of the data model from the previous sections.[1] Let us assume that the associations attends and belongsTo are navigable in both directions, resulting in two opposed edges between Documents and Lectures and between Lectures and Students.

Figure 5.5: Sample graph representation

5.4.2 Graph Representation of System Instance

A similar representation is introduced for system instances, called the *instance graph* $g = (O, L)$ consisting of vertices $O = \{o_1, ..., o_k\}$, representing objects, and edges $L = \{l_1, ..., l_l\}$ representing links between objects. For every object, a vertex labeled with the fully qualified object name (i.e. object name, class name and class tags) has

[1] To illustrate that the figures represent a graph model, and not one of the data- or instance graphs from section 5.1, we use a different graphical notation.

to be added to the instance graph. For every link between two objects, i.e. for every typed attribute value

$$\text{[sourceObject]:[sourceClass].[associationName]} = \text{[targetObject]:[targetClass]}$$

or

$$\text{[sourceObject]:[sourceClass].[associationName]} = \{\text{[o1]:[targetClass],...,[oN]:[targetClass]}\}$$

for $1:n$ and $m:n$ associations, an edge from [sourceObject] to [targetObject] is added to the graph. This linking edge is labeled with the association name.

Figure 5.6 shows the graph representation corresponding to the system instance from the previous sections.

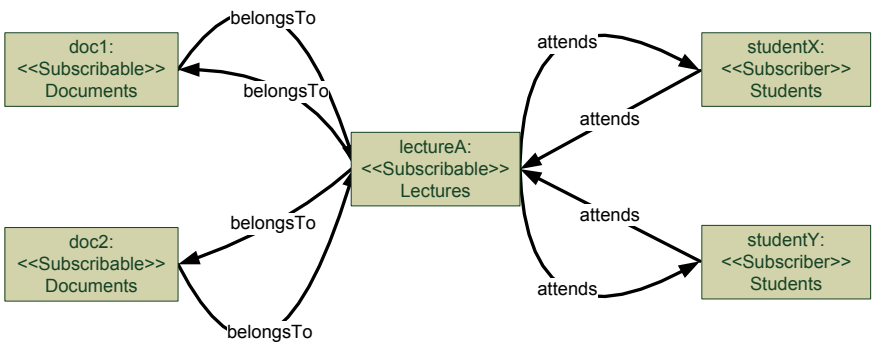


Figure 5.6: Sample system instance graph representation

5.4.3 Graph Representation of Overlays

Given an overlay $\mathcal{O}_i = (\text{[className]}, A_{implicitSub}, A_{eventProp})$, the corresponding graph representation $G^{\mathcal{O}_i} = (C^{\mathcal{O}_i}, A^{\mathcal{O}_i})$ consists of edges for all implicit subscription associations $A_{implicitSub}$ and event-propagating associations $A_{eventProp}$, together with the respective tags. Additionally, all adjacent vertices of the data model graph are added to the overlay graph.

Figure 5.7 shows the graph representation of overlay $\mathcal{O}_1$ of the example in section 5.3.

Figure 5.7: Sample overlay graph representation

5.4.4 Path Descriptions

Path descriptions are used to describe a path along classes in the graph representation of the data model. In the following, we will explain how they are defined.

Definition 5.4.1 (Path Descriptions) Let $a_1, ..., a_n$ be associations in a model graph. The tuple

$$p = (a_1, ..., a_n)$$

is called a *path description* if it represents a connected path in the data model, i.e. for every

$$i \in [1, n-1],$$

$$a_i = \text{[sourceClassA].[associationNameA]:[targetClassA]}$$

$$\textit{and}$$

$$a_{i+1} = \text{[sourceClassB].[associationNameB]:[targetClassB]}$$

we demand that

$$\text{[targetClassA]} = \text{[sourceClassB]}.$$

A sample path description p in the data model of figure 5.5 is

$$p = (\ll\text{Subscribable}\gg\text{Documents.belongsTo:}\ll\text{Subscribable}\gg\text{Lectures},$$

$$\ll\text{Subscribable}\gg\text{Lectures.attends:}\ll\text{Subscriber}\gg\text{Students}).$$

Although path descriptions contain only associations, the corresponding class vertices along the path can be inferred from the association sources and targets.

5.4.5 Path Instances

A *path instance* denotes the equivalent of path descriptions in an instance graph.

Definition 5.4.2 (Path Instances) Let $l_1, ..., l_m$ be links in a system instance graph. The tuple

$$pi = (l_1, ..., l_m)$$

is called *path instance*, if it represents a connected set of links in the system instance, i.e. if for every

$$j \in [1..m-1],$$

$$pi_j = \text{[sourceObjectA]:[sourceClassA].[associationNameA]} = \text{[targetObjectA]}$$

and

$$pi_{j+1} = \text{[sourceObjectB]:[sourceClassB].[associationNameB]} = \text{[targetObjectB]},$$

$$\text{[targetObjectA]} = \text{[sourceObjectB]}$$

holds.

Figure 5.6 contains several path instances, e.g.

$$pi =$$

$$(\text{doc1:}\ll\text{Subscribable}\gg\text{documents.}\ll\text{eventProp}\gg\text{belongsTo} = \text{lectureA:}\ll\text{Subscribable}\gg\text{Lectures},$$

$$\text{lectureA:}\ll\text{Subscribable}\gg\text{Lectures.}\ll\text{implicitSub}\gg\text{attends} = \text{studentY:}\ll\text{Subscriber}\gg\text{Students}).$$

Definition 5.4.3 (Matching Path Descriptions and Path Instances) Given a path description $p = (a_1, ..., a_n)$ and a path instance $pi = (l_1, ..., l_m)$ with

$$a_i = \text{[sourceClassI].[associationNameI]:[targetClassI]}$$

and

$$l_j = \text{[sourceObjectJ]:[sourceClassJ].[instanceAssociationNameJ]} = \text{[targetObjectJ]}$$

we say that pi *matches* p, denoted by

$$pi \subseteq_{matches} p,$$

if and only if

$$m = n$$

and

$$\forall i \in [1..n] : [associationNameI] = [instanceAssociationNameI].$$

Given the path description

$$p = (\mathsf{Documents.belongsTo{:}Lectures}, \mathsf{Lectures.attends{:}Students})$$

from figure 5.5 and the two path instances[2]

$$pi_1 = (\mathsf{doc1{:}Documents.belongsTo} = \mathsf{lectureA{:}Lectures},$$

$$\mathsf{lectureA{:}Lectures.attends} = \mathsf{studentX{:}Students})$$

and

$$pi_2 = (\mathsf{doc1{:}Documents.belongsTo} = \mathsf{lectureA{:}Lectures},$$

$$\mathsf{lectureA{:}Lectures.belongsTo} = \mathsf{doc2{:}Documents}),$$

this means that

$$pi_1 \subseteq_{matches} p$$

and

$$\neg(pi_2 \subseteq_{matches} p).$$

5.5 Interpretation

In the following, we will explain how to interpret our event-handling constructs. First, we have to introduce several helper constructs.[3]

5.5.1 Definitions

Definition 5.5.1 (Event-Propagating Path Descriptions) Let $p = (a_1, ..., a_n)$ be a path description in the model graph G, let

$$\mathcal{O}_x = ([\mathsf{sourceClass}], A_{implicitSub}, A_{eventProp})$$

be an arbitrary overlay. We say that p is an *event-propagating path description* in $\mathcal{O}_x$ iff

$$\forall j \in [1..n] : a_j \in A_{eventProp}.$$

[2]For better readability, tags have been omitted in the path description and the path instances.

[3]From now on, we abbreviate the names of classes, attributes and assocations with [...] whenever they are not relevant for the current definition or explanation.

Definition 5.5.2 (Association Count Function) Let $p = (a_1, ..., a_n)$ be a path description in the model graph G. The association count function f_{ac} counts the occurrences of a particular association [associationName] in this path, i.e.

$$f_{ac}([associationName], p) :=$$

$$|\{a_i | a_i = [\ldots].[\text{associationName}]:[\ldots]\}|$$

Definition 5.5.3 (Valid Event-Propagating Path Descriptions) Let

$$p = (a_1, ..., a_n)$$

be a path description in the model graph G. We call p a *valid event-propagating path description*, if and only if p is an event-propagating path description (cf. definition 5.5.1) and

$$\forall i \in [1..n]$$

with

$$a_i = [\ldots].\ll\text{eventProp @impactRange=r}\gg[\text{associationName}]:[\ldots]$$

a maximum of r associations are contained in p, i.e.

$$f_{ac}([associationName], p) \leq r$$

holds.

Example To illustrate the definitions introduced above, let us take a look at the model graph shown in figure 5.8.

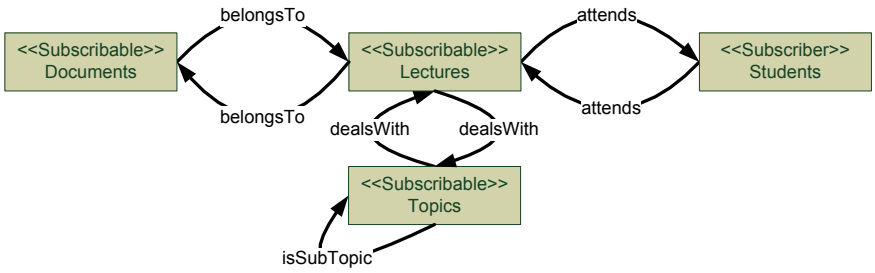


Figure 5.8: Sample graph representation to illustrate path definitions

Let us further consider the overlay

$$\mathcal{O}_i = (\text{Documents}, \{\text{Documents.belongsTo:Lectures}, \text{Lectures.dealsWith:Topics},$$

$$\mathsf{Topics}.\ll\mathsf{eventProp\ @impactRange{=}2}\gg\mathsf{isSubTopic{:}Topics}\},$$

$$\{\mathsf{Lectures.attends{:}Students}\}).$$

In this scenario, let us take a look at three different path descriptions. We start with path description

$$p_1 = (\mathsf{Documents.belongsTo{:}Lectures}, \mathsf{Lectures.attends{:}Students}).$$

Since the second association attends is no event-propagating association, definition 5.5.1 is not fulfilled, i.e. p_1 is no event-propagating path description and thus (cf. def. 5.5.3) no valid event-propagating path description either.

Looking at path description

$$p_2 = (\mathsf{Documents.belongsTo{:}Lectures}, \mathsf{Lectures.dealsWith{:}Topics},$$

$$\mathsf{Topics.isSubTopic{:}Topics}, \mathsf{Topics.isSubTopic{:}Topics}, \mathsf{Topics.isSubTopic{:}Topics})$$

we can state that all contained associations are event-propagating, so definition 5.5.1 holds. Association belongsTo has no explicit impact range, i.e. the default impact range 1 is used. Since p_2 contains exactly one instance of belongsTo, the requirement of definition 5.5.3 is not violated. The same holds for dealsWith. However, 3 instances of isSubTopic are contained in p_2, while the maximum impact range is defined to be 2, so p_2 is an event-propagating path description, but not a valid event-propagating path description with respect to definition 5.5.3.

Considering the last path description

$$p_3 = (\mathsf{Documents.belongsTo{:}Lectures}, \mathsf{Lectures.dealsWith{:}Topics},$$

$$\mathsf{Topics.isSubTopic{:}Topics}, \mathsf{Topics.isSubTopic{:}Topics}),$$

only two instances of association isSubtopic are present, so definition 5.5.3 also holds and p_3 thus is a valid event-propagating path description.

5.5.2 Subscribers to Inform about Updates

Based on the above definitions, the semantically correct subscribers that have to be notified about updates within an overlay are defined as follows:

5.5.2.1 Implicit Subscribers

Implicit subscribers are defined as follows:

Definition 5.5.4 (Implicit Subscribers) Let g be an arbitrary instance graph, let

$$\mathcal{O}_i = (\text{[sourceClass]}, A_{implicitSub}, A_{eventProp})$$

be an arbitrary overlay as defined in def. 5.3.1. Let $PI = \{pi_1, ..., pi_n\}$ be the set of all path instances in g. If an update on an instance

[subscribableObject]:[sourceClass]

is detected, an object

[targetObject]:[targetClName]

contained in g has to be informed about this update if there exists a path instance $pi_i = (l_1, ..., l_m) \in PI$ fulfilling the following requirements:

1. there exists at least one valid event-propagating path description in $G^{\mathcal{O}_i}$ matching the sub-path $(l_1, ..., l_{m-1})$, i.e.

   $$\exists p : p \text{ is a valid event-propagating path description and} (l_1, ..., l_{m-1}) \subseteq_{matches} p$$

2. the instance path starts from the monitored subscribable, i.e.

   $$l_1 = \text{[subscribableObject]:}\ll\text{Subscribable}\gg\text{[sourceClass].[...]} = \text{[...]:[...]}$$

3. the last link in the instance path is an implicit subscription[4] targeting at [targetClName], i.e.

   $$l_m = \text{[...]:}\ll\text{Subscribable}\gg\text{[...].}\ll\text{implicitSub}\gg\text{[...]} = \text{[targetObject]:[targetClNameM]}$$

The set of all implicit subscribers that have to be informed about an update of [subscribableObject]:[sourceClass] is denoted by

$$C^{implicit}_{Subscriber}(\text{[subscribableObject]}, \mathcal{O}_i).$$

Informally spoken, all implicit subscribers of an event-propagation starting from the monitored subscribable have to be notified implicitly.

[4] In section 5.7.2 we will discuss why implicit subscriptions are *not* transitive and thus can only be the last association link.

5.5.2.2 Explicit Subscribers

Similarly, explicit subscribers are defined:

Definition 5.5.5 (Explicit Subscribers) Let g be an arbitrary instance graph, let

$$\mathcal{O}_i = (\text{[sourceClass]}, A_{implicitSub}, A_{eventProp})$$

be an arbitrary overlay as defined in def. 5.3.1. Let $PI = \{pi_1, ..., pi_n\}$ be the set of all path instances in g. If a modification of [subscribableObject]:[sourceClass] is detected, an object [targetObject]:[targetCIName] contained in the instance graph g has to be informed about this update, if there exists a path instance $pi_i = (l_1, ..., l_m) \in PI$ fulfilling the following requirements:

1. there exists at least one valid event-propagating path description $p \in G^{\mathcal{O}_i}$ matching pi_i, i.e.

   $$\exists p : p \text{ is a valid event-propagating path description and } pi_i \subseteq_{matches} p$$

2. pi_i starts from the monitored subscribable, i.e.

   $$l_1 = \text{[subscribableObject]:}\ll\text{Subscribable}\gg\text{[sourceClass].[...]} = \text{[...]:[...]}$$

3. there exists an explicit subscription between the target of the event-propagation and [targetObject]:[targetCIName], i.e. for

   $$l_m = \text{[...]:[...].[...]} = \text{[targetObjectM]:[targetClassM]}$$

   there has to be an explicit subscription

   $$exp \in Sub_{explicit}$$

   with

   $$exp = (\text{[targetObjectM]:[targetClassM]}, \text{[targetObject]:[targetCIName]}).$$

The set of all explicit subscribers that have to be informed about an update of [subscribableObject]:[sourceClass] is denoted by

$$C^{explicit}_{Subscriber}(\text{[subscribableObject]}, \mathcal{O}_x)$$

This definition states that all explicit subscribers of an event-propagation starting from [subscribableClName] have to be notified explicitly.

An object [subscriberObject]:[subscriberClass] has to be notified if it either has to be notified explicitly *or* implicitly:

Definition 5.5.6 (Subscribers to Notify in an Overlay) Let g be an arbitary instance graph, let

$$\mathcal{O}_i = ([\mathsf{sourceClass}], A_{implicitSub}, A_{eventProp})$$

be an arbitrary overlay as defined in def. 5.3.1. If an update on [subscribableObject]:[sourceClass] is detected, all subscribers that have to be notified about this update are

$$C^{all}_{Subscriber}([\mathsf{subscribableObject}], \mathcal{O}_i) :=$$
$$C^{implicit}_{Subscriber}([\mathsf{subscribableObject}], \mathcal{O}_i) \cup C^{explicit}_{Subscriber}([\mathsf{subscribableObject}], \mathcal{O}_i)$$

Finally, the subscribers that have to be notified for every overlay are united to get the overall set of subscribers:

Definition 5.5.7 (Subscribers to Notify about an Update) Let

$$\mathcal{O} = \{\mathcal{O}_1, ..., \mathcal{O}_n\}$$

be the set of all overlays, let [sourceObject] be an updated object. The subscribers that have to be notified are

$$C^{all}_{Subscriber}([\mathsf{sourceObject}], \mathcal{O}) := \cup_{i \in [1..n]} C^{all}_{Subscriber}([\mathsf{sourceObject}], \mathcal{O}_i)$$

5.6 Real-Life Example

Next, we will present a real-life example together with its formal specification, and show how to interpret this specification in case of an update.

5.6.1 Sample Scenario

As an example, we use a data model derived from figure 5.1 that has been slightly modified: instead of storing the room as an attribute of lectures, we introduce a new

entity **Rooms** which is associated to lectures. Additionally, rooms can be hierarchically organized, i.e. rooms (or buildings) can contain each other. A room may have a maintainer who is responsible for its maintenance. Figure 5.9 shows the data model.

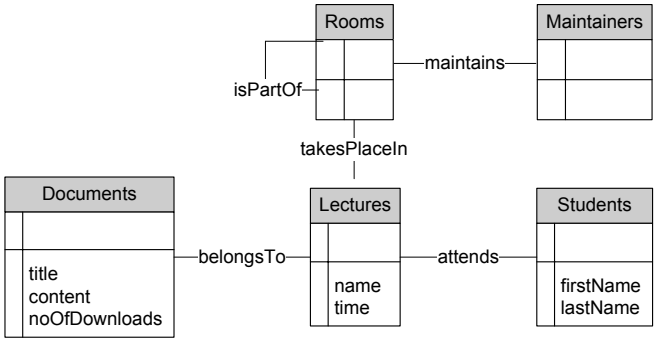


Figure 5.9: Real-life data model

Assume that the following requirements have been specified:

- Whenever the contents of a document are updated, all students attending a lecture referring to this document have to be informed.
- Whenever a lecture is re-scheduled to a different starting time, the room's maintainer and all maintainers of its parent rooms/buildings (up to two levels of hierarchy above) have to be informed.

Figures 5.10 and 5.11 show these two use cases graphically.

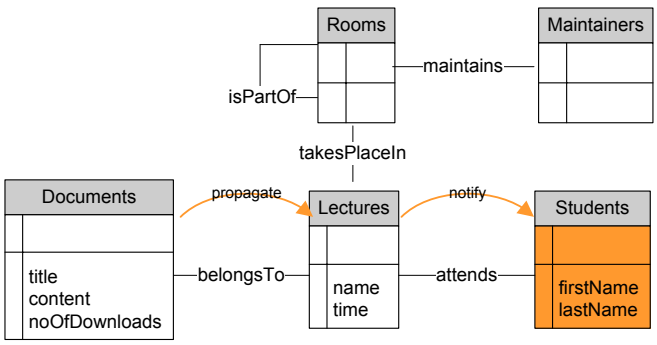


Figure 5.10: First overlay

Let us take a look at the instance representation shown in figure 5.12, containing several students, documents, lectures, rooms and maintainers. For better readability,

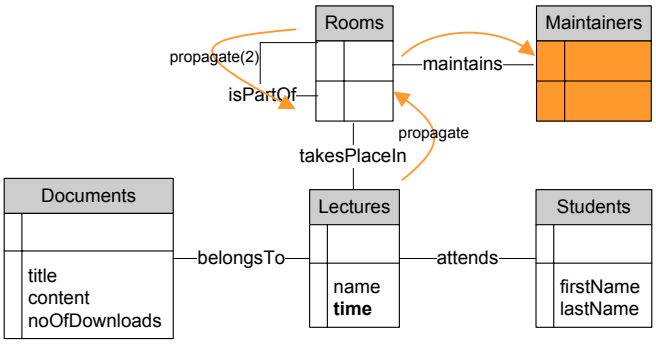


Figure 5.11: Second overlay

we do not consider the attribute values, since they are not needed for our explanation. Classnames are also omitted for better readability.

5.6.2 Formal Representation

Represented using our formalism, the scenario can be expressed as follows. The data model contains the following classes, attributes and associations:

- Class Documents with attributes title:String, content:URL, noOfDownloads:Integer and belongsTo:Lectures,

- class Lectures with attributes name:String, time:Time, belongsTo:Documents, takesPlaceIn:Rooms and attends:Students,
- class Rooms with attributes isPartOf:Rooms, maintains:Maintainers and takesPlaceIn:Lectures,
- class Students with attributes firstName:String, lastName:String and attends:Lectures, and finally
- class Maintainers with attribute maintains:Rooms.

Graphically, this is represented by figure 5.13. The event-handling constructs from the first overlay (figure 5.10) monitoring ≪Subscribable≫Documents are expressed as

≪Subscribable≫Documents
≪Subscribable≫Lectures

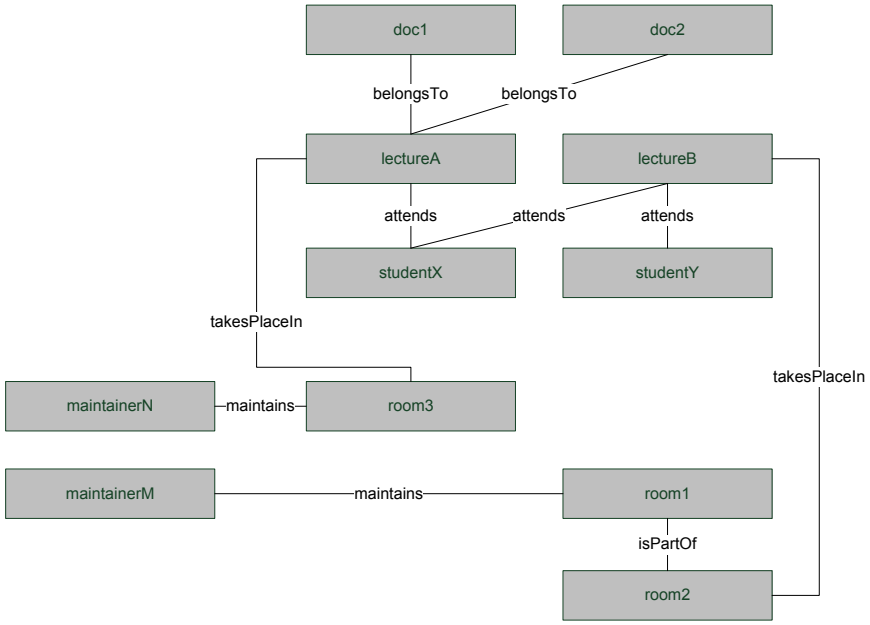


Figure 5.12: Sample instance graph

≪Subscriber≫Students

≪Subscribable≫Documents.≪Subscribable≫content
≪Subscribable≫Documents.≪eventProp≫belongsTo:≪Subscribable≫Lectures
≪Subscribable≫Lectures.≪implicitSub≫attends:≪Subscriber≫Students

The second overlay (figure 5.11) monitoring ≪Subscribable≫Lectures is similarly expressed as

≪Subscribable≫Lectures
≪Subscribable≫Rooms
≪Subscriber≫Maintainers

≪Subscribable≫Lectures.≪Subscribable≫time
≪Subscribable≫Lectures.≪eventProp≫takesPlaceIn:≪Subscribable≫Rooms
≪Subscribable≫Rooms.≪eventProp @impactRange = 2≫isPartOf:≪Subscribable≫Rooms
≪Subscribable≫Rooms.≪implicitSub≫maintains:≪Subscriber≫Maintainers

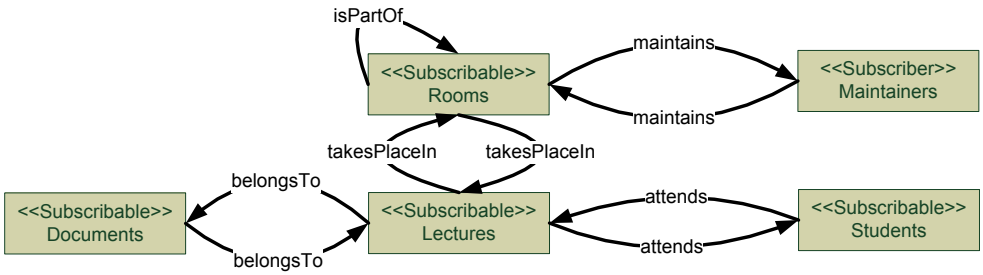


Figure 5.13: Graph representation of data model

To complete our scenario, we add an explicit subscription between lectureA and maintainerM:

$$Sub_{explicit} = \{(\mathsf{lectureA}, \mathsf{maintainerM})\}$$

5.6.3 Interpretation of the Event-Handling Specification

In the graph based representation, the two overlays from figure 5.10 and 5.11 are visualized in figure 5.14 and 5.15.

Figure 5.14: Graph representation of first overlay

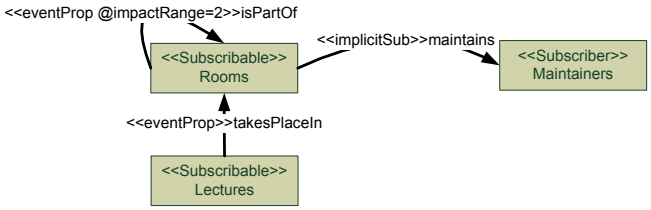


Figure 5.15: Graph representation of second overlay

Overlay $\mathcal{O}_1$ Since there is only one event-propagating association, the first overlay contains only the following valid event propagating path description:

$$p_{1,1} = (\ll\text{Subscribable}\gg\text{Documents}.\ll\text{eventProp}\gg\text{belongsTo}:\ll\text{Subscribable}\gg\text{Lectures})$$

In the instance model (cf. figure 5.12) , the following set of path instances matches this path description (we omit the class names for better readability):

$$PI_1 = \{(\text{doc1.belongsTo} = \text{lectureA}), (\text{doc2.belongsTo} = \text{lectureA})\}$$

The only implicit subscription in $\mathcal{O}_1$ is attends, so only the following path description satisfies the requirement from definition 5.5.4:

$$p^i_{1,1} = (\ll\text{Subscribable}\gg\text{Documents}.\ll\text{eventProp}\gg\text{belongsTo}:\ll\text{Subscribable}\gg\text{Lectures},$$
$$\ll\text{Subscribable}\gg\text{Lectures}.\ll\text{implicitSub}\gg\text{attends}: \ll\text{Subscriber}\gg\text{Students})$$

The following path instances match this description:

$$PI^i_1 = \{(\text{doc1.belongsTo} = \text{lectureA}, \text{lectureA.attends} = \text{studentX}),$$
$$(\text{doc2.belongsTo} = \text{lectureA}, \text{lectureA.attends} = \text{studentX})\}$$

Overlay $\mathcal{O}_2$ Due to the reflexive event-propagating association isPartOf, the second overlay contains an infinite number of event-propagating path descriptions, since every path description containing class Rooms can always be extended by an additional association edge isPartOf. However, the impact range of 2 limits the allowed number of those associations in a valid event-propagating path description, so that only the following valid event-propagating path descriptions remain:

$$p_{2,1} = (\ll\text{Subscribable}\gg\text{Lectures}.\ll\text{eventProp}\gg\text{takesPlaceIn}:\ll\text{Subscribable}\gg\text{Rooms})$$
$$p_{2,2} = (\ll\text{Subscribable}\gg\text{Lectures}.\ll\text{eventProp}\gg\text{takesPlaceIn}:\ll\text{Subscribable}\gg\text{Rooms},$$
$$\ll\text{Subscribable}\gg\text{Rooms}.\ll\text{eventProp}\gg\text{isPartOf}:\ll\text{Subscribable}\gg\text{Rooms})$$
$$p_{2,3} = (\ll\text{Subscribable}\gg\text{Lectures}.\ll\text{eventProp}\gg\text{takesPlaceIn}:\ll\text{Subscribable}\gg\text{Rooms},$$
$$\ll\text{Subscribable}\gg\text{Rooms}.\ll\text{eventProp}\gg\text{isPartOf}:\ll\text{Subscribable}\gg\text{Rooms},$$
$$\ll\text{Subscribable}\gg\text{Rooms}.\ll\text{eventProp}\gg\text{isPartOf}:\ll\text{Subscribable}\gg\text{Rooms})$$

Given the instance from figure 5.12, the following paths match one of these descriptions each:

$$PI_2 = \{(\text{lectureA.takesPlaceIn} = \text{room3}),$$
$$(\text{lectureB.takesPlaceIn} = \text{room2, room2.isPartOf} = \text{room1})\}$$

Due to the only implicit subscription maintains in $\mathcal{O}_2$, the following path descriptions satisfy the requirement from definition 5.5.4:

$$p^i_{2,1} = (\ll\text{Subscribable}\gg\text{Lectures.}\ll\text{eventProp}\gg\text{takesPlaceIn:}\ll\text{Subscribable}\gg\text{Rooms},$$
$$\ll\text{Subscribable}\gg\text{Rooms.}\ll\text{implicitSub}\gg\text{maintains:}\ll\text{Subscriber}\gg\text{Maintainers})$$

$$p^i_{2,2} = (\ll\text{Subscribable}\gg\text{Lectures.}\ll\text{eventProp}\gg\text{takesPlaceIn:}\ll\text{Subscribable}\gg\text{Rooms},$$
$$\ll\text{Subscribable}\gg\text{Rooms.}\ll\text{eventProp}\gg\text{isPartOf:}\ll\text{Subscribable}\gg\text{Rooms},$$
$$\ll\text{Subscribable}\gg\text{Rooms.}\ll\text{implicitSub}\gg\text{maintains:}\ll\text{Subscriber}\gg\text{Maintainers})$$

$$p^i_{2,3} = (\ll\text{Subscribable}\gg\text{Lectures.}\ll\text{eventProp}\gg\text{takesPlaceIn:}\ll\text{Subscribable}\gg\text{Rooms},$$
$$\ll\text{Subscribable}\gg\text{Rooms.}\ll\text{eventProp}\gg\text{isPartOf:}\ll\text{Subscribable}\gg\text{Rooms},$$
$$\ll\text{Subscribable}\gg\text{Rooms.}\ll\text{eventProp}\gg\text{isPartOf:}\ll\text{Subscribable}\gg\text{Rooms},$$
$$\ll\text{Subscribable}\gg\text{Rooms.}\ll\text{implicitSub}\gg\text{maintains:}\ll\text{Subscriber}\gg\text{Maintainers})$$

The following path instances match one of these descriptions:

$$PI^i_2 = \{(\text{lectureA.takesPlaceIn} = \text{room3, room3.maintains} = \text{maintainerN}),$$
$$(\text{lectureB.takesPlaceIn} = \text{room2, room2.isPartOf} = \text{room1, room1.maintains} = \text{maintainerM})\}$$

Let us take a look at three different possible kinds of updates:

Update of attribute title of a document Since title is not a monitored attribute in any of the two overlays, no event handling is triggered at all.

Update of attribute content of doc1 First of all, content is a monitored attribute of Documents in overlay $\mathcal{O}_1$ only, so the detected update has to be handled according to $\mathcal{O}_1$. As we already showed, PI_1 contains the following path instance starting with doc1: (doc1.belongsTo = lectureA) and PI_1^i contains the following path instance starting with doc1:

$$(\mathsf{doc1.belongsTo} = \mathsf{lectureA}, \mathsf{lectureA.attends} = \mathsf{studentX}).$$

To determine the explicit subscribers $C^{explicit}_{Subscriber}([doc1], \mathcal{O}_1)$, all explicit subscriptions between the path instance targets in PI_1, i.e. lectureA, and any subscriber have to be found. Obviously, there is only the explicit subscription (lectureA, maintainerM), so that

$$C^{explicit}_{Subscriber}(\mathsf{doc1}, \mathcal{O}_1) = \{\mathsf{maintainerM}\}.$$

Additionally, all implicit subscribers, i.e. the targets of PI_1^i have to be determined:

$$C^{implicit}_{Subscriber}(\mathsf{doc1}, \mathcal{O}_1) = \{\mathsf{studentX}\}.$$

Finally, all subscribers to notify are

$$C^{all}_{Subscriber}(\mathsf{doc1}, \mathcal{O}_1) = C^{implicit}_{Subscriber}(\mathsf{doc1}, \mathcal{O}_1) \cup C^{explicit}_{Subscriber}(\mathsf{doc1}, \mathcal{O}_1) =$$

$$= \{\mathsf{studentX}\} \cup \{\mathsf{maintainerM}\} = \{\mathsf{studentX}, \mathsf{maintainerM}\}.$$

Update of attribute title of lectureB Lecture is a monitored subscribable in $\mathcal{O}_2$ only. The targets of the paths in PI_2 starting with lectureB are room2 and room1. However, there are no explicit subscriptions between a subscriber and one of these targets, so

$$C^{explicit}_{Subscriber}(\mathsf{lectureB}, \mathcal{O}_2) = \emptyset.$$

To determine the implicit subscribers, we look at all paths in PI_2^i starting with lectureB and get maintainerM as a result, so

$$C^{explicit}_{Subscriber}(\mathsf{lectureB}, \mathcal{O}_2) = \{\mathsf{maintainerM}\}.$$

Finally, all subscribers to notify are

$$C^{all}_{Subscriber}(\mathsf{lectureB}, \mathcal{O}_2) = C^{implicit}_{Subscriber}(\mathsf{lectureB}, \mathcal{O}_2) \cup C^{explicit}_{Subscriber}(\mathsf{lectureB}, \mathcal{O}_2) =$$

$$= \{\mathsf{maintainerM}\} \cup \emptyset = \{\mathsf{maintainerM}\}.$$

5.7 Discussion on Design Decisions

The design decisions we made when developing the above-mentioned concept were not always absolutely obvious. To answer questions which readers might ask, we pick a few aspects of our concept and explain why they were designed as they are in the following.

5.7.1 Handling Inheritance

Although not being explicitly covered by our concept, inheritance has to be implicitly handled by the event-handling component according to the following rules:

- If superclass Super is tagged as subscribable, any subclass Sub is subscribable, too.
- If superclass Super is tagged as subscriber, any subclass Sub is a subscriber, too.
- If superclass Super is the source or the target of an event-propagating association or an implicit subscription, all subclasses are source / target of the respective association, too.

Not all kinds of data models support inheritance. For instance, entity-relationship diagrams offer no explicit possibility to define superclass-subclass relations. Thus, it is the responsibility of the concrete implementation of our approach to correctly support inheritance, if the data model contains super- and subclasses. Although this is not part of our work, we will outline how inheritance can be supported when using UML models and relational database triggers in part III.

5.7.2 Non-Transitive Implicit Subscriptions

In our concept, event-propagating references are transitive, while implicit subscriptions are not. This has been designed like this on purpose: first of all, according to our semantic understanding, an implicit subscription should always be a consequence of the fact that the update of a *subscribable* A leads to a notification to the adjacent *subscriber* B. Thus, B can not be the source of a second, transitive implicit subscription. Second, a transitive implicit subscription between *subscribers* would mean that subscribers notify other subscribers about update events, which is not what we want to express with

implicit subscriptions. In this case, we would rather recommend to use entities that group users (cf. *organizational units* in LDAP) and define those groups as subscribers.

However, there are imaginable scenarios where semantics that resemble transitive implicit subscriptions might be needed. Let us assume the information system contains a family tree with persons and their children or ancestors, respectively. Let us further assume that the event-handling semantics we want to model are: "If information about a direct or indirect ancestor is modified, all children have to be informed implicitly."

Instead of requiring transitive implicit subscriptions, this requirement can be fulfilled by using an overlay specification as depicted in figure 5.16.

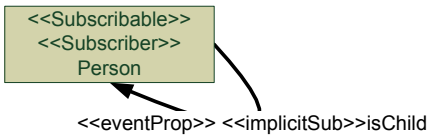


Figure 5.16: Sample overlay for family tree notification

Class Person is tagged both as ≪Subscribable≫ and ≪Subscriber≫ and the reflexive association isChild is designed to be ≪implicitSub≫ *and* ≪eventProp≫. Thus, every update of a person automatically transitively marks all children as updated, too, and whenever a person is updated, its children are implicitly notified. Thus, all children of an updated person are automatically implicitly notified, even if implicit subscriptions are not transitive themselves.

5.7.3 Benefit of Overlays

Due to the separate evaluation of overlays, event-propagation within one overlay can **not** affect any other overlay. This behaviour is intended, since every overlay is designed to handle exactly one particular processing instruction for updates.

However, there is one situation in which this behaviour is not helpful: if several subscribables have to be monitored and handled according to the same processing instruction, i.e. they have to share the same implicit subscriptions and event-propagating associations, this intention can not be directly modelled using our approach. Instead of specifying only one subscribable as the source of an overlay, a whole set of subscribables would be required.

Although this is not possible in our solution, a simple workaround solves the problem: by specifying multiple overlays, each of them having a different source subscribable but the same implicit subscriptions and event-propagating associations, the intended behaviour can be realized.

Formally, one would expect that for a multi-source overlay

$$\mathcal{O}_{multi} = (\{[\mathsf{sourceCl1}], ..., [\mathsf{sourceCln}]\}, A_{implicitSub}, A_{eventProp})$$

and for several overlays

$$\mathcal{O}_i = ([\mathsf{sourceCli}], A_{implicitSub}, A_{eventProp})$$

the notification semantics is to be defined as

$$C^{all}_{Subscriber}([\mathsf{sourceObject}], \mathcal{O}_{multi}) :=$$
$$C^{all}_{Subscriber}([\mathsf{sourceObject}], \mathcal{O}_1) \cup ... \cup C^{all}_{Subscriber}([\mathsf{sourceObject}], \mathcal{O}_n)$$

Since the final result, considering all overlays, is the union of the subscribers determined when interpreting every overlay individually (cf. def. 5.5.7), the same effect can also be obtained by specifying several separate overlays, one per source class each.

5.8 Summary

In this chapter, we presented the semantic concepts and a formal model as a foundation of the *data model* and *event model* of our approach. Using these two models, we explained how to interpret these specifications and determine which subscribers to notify about updates. In the following chapter, we will introduce a generic algorithm taking this specification as an input and derive an implementation that implements the formal concept, thus realizing the event-handling component on an abstract level.

"I have yet to see any problem, however complicated, which, when you looked at it in the right way, did not become still more complicated."

Poul Anderson

6 Generic Trigger Generation

In this chapter, we show how to generically generate triggers[1] for an information system, taking the event-handling specification, which has been introduced in the previous chapter, as an input. The generation process is independent of the actual information system implementation, i.e. we do not rely on particular implementation techniques like e.g. relational database triggers. The generation algorithm itself is formally presented and explained using a running example. The correctness of the algorithm is proved. The results from this chapter thus serve as a basis for an actual implementation. A sample implementation using active database technology will be presented in part III of this dissertation.

6.1 Overview of the Generation Algorithm

Figure 6.1 shows the single steps within the overall generation process, classified according to the different layers that are touched: the modelling layer, the event-handling

[1] In the following, we will use the term "trigger" for the executable code that monitors updates in the information system and determines all subscribers to notify. Triggers, in that sense, are not necessarily triggers as known from active database systems, but can be realized using any technique that is suitable for the underlying information system.

system that has to be created and the runtime information system containing the data that will be monitored.

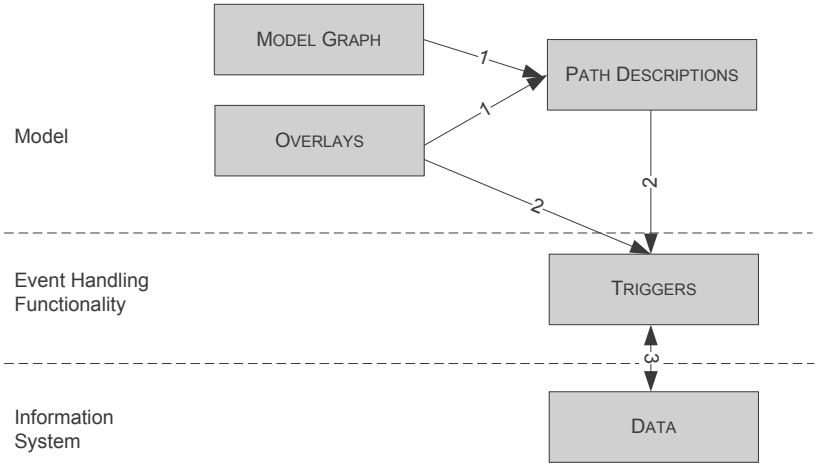


Figure 6.1: Generic generation procedure

In a first step, the specifications of the data model and of the overlays are taken, and a set of path descriptions (cf. def. 5.4.1) are computed. These path descriptions are then used in conjunction with the overlay information to generate triggers that realize the event-handling functionality. At runtime, these triggers then monitor data modifications in the actual information system.

6.2 Generation Algorithm in Detail

The overall generation procedure takes a set of overlays and a graph representation of the data model as an input and returns a set of triggers which realize the event-handling component. On an abstract level, we define a trigger as follows:

Definition 6.2.1 (Triggers) Let [monitoredClass] $\in C$ be a class in the data model, let $Q = \{q_1, \ldots, q_n\}$ be a set of selection queries, taking an object [input] $\in O$ as an input, so that for each $q_1 \in Q$ the evaluation of the query

$$eval(q_i, [\mathsf{input}]) = \{r_1, ..., r_m\}, r_i \in O$$

returns a set of resulting objects r_j, representing all subscribers that result from the update of [input]. A trigger

$$t = ([\text{monitoredClass}], Q)$$

detects modifications of instances *input* of class [monitoredClass] in the information system and returns the results of each query in case such a modification is detected. The results are the union of the evaluation of each contained query and denoted by

$$eval([\text{input}], t) = eval([\text{input}], q_1) \cup \ldots \cup eval([\text{input}], q_n).$$

Algorithm 6.1 shows how these triggers are created.

Algorithm 6.1: GenerateTriggers

Input: Set of overlays $\mathcal{O} = \{\mathcal{O}_1, \ldots, \mathcal{O}_n\}$
Output: Set of triggers $T = \{t_1, \ldots, t_m\}$

1 $T \leftarrow \emptyset$;
2 **foreach** $\mathcal{O}_i = ([\textit{sourceClass}], A_{implicitSub}, A_{eventProp}) \in \mathcal{O}$ **do**
3 $\quad ExpPathDesc \leftarrow ComputeExplicitPathDescriptions(\mathcal{O}_i)$;
4 $\quad ImpPathDesc \leftarrow ComputeImplicitPathDescriptions(\mathcal{O}_i)$;
5 $\quad T \leftarrow T \cup CreateExplicitTriggers(ExpPathDesc)$;
6 $\quad T \leftarrow T \cup CreateImplicitTriggers(ImpPathDesc)$;
7 **end**

For every overlay, a set of explicit and implicit path descriptions is computed (cf. algorithms 6.2 and 6.3). Then, triggers for the implicit path descriptions and for the explicit path descriptions are created, monitoring the subscribable of the overlay and containing queries as computed by algorithms 6.5 and 6.6.

To generate the explicit and implicit path descriptions, the algorithms 6.2 and 6.3 are used. Both of them get an overlay as input and create a set of path descriptions as introduced in definition 5.4.1.

The algorithm to create the explicit path descriptions simply returns the set of all valid event-propagating path descriptions (cf. def. 5.5.3), which are computed according to algorithm 6.4. This result will later be processed by the generation of explicit triggers (alg. 6.5) and implicit triggers (alg. 6.6).

Algorithm 6.2: ComputeExplicitPathDescriptions

Input: Overlay $\mathcal{O}_i = ([\mathsf{sourceClass}], A_{implicitSub}, A_{eventProp})$
Output: Set of path descriptions $P = \{p_1, \ldots, p_n\}$

1 $P \leftarrow ComputeValidEventPropagatingPathDescriptions(\mathcal{O}_i)$;

The computation of implicit path descriptions is also based on the valid event-propagating path descriptions in an overlay. For every valid event-propagating path description that ends with a subscribable class, one path description per implicit subscription is added to the result, i.e., the original description is extended by the implicit subscription association. This procedure is shown in algorithm 6.3.

Algorithm 6.3: ComputeImplicitPathDescriptions

Input: Overlay $\mathcal{O}_i = ([\mathsf{sourceClass}], A_{implicitSub}, A_{eventProp})$
Output: Set of implicit path descriptions $P = \{p_1, \ldots, p_n\}$

1 $eventProp \leftarrow ComputeValidEventPropagatingPathDescriptions(\mathcal{O}_i)$;
2 **foreach** $(a_1, \ldots, a_m) \in eventProp$ *with* $a_m = [A].[x] = [B]$ **do**
3 // find implicit subscriptions starting from the last class in the
4 // valid event-propagating path description
5 **foreach** $a_{imp} \in A_{implicitSub}$ *with* $a_{imp} = [B].[y] = [C]$ **do**
6 // extend this path description with the implicit subscription
7 $P \leftarrow P \cup (a_1, \ldots, a_m, a_{imp})$;
8 **end**
9 **end**

Next, we show how the valid event-propagating path descriptions are computed. This is done using a modified version of depth-first search, formally shown in algorithm 6.4.

Starting with an arbitrary event-propagating path description *current* and the currently last subscribable in this path description [currentTarget], all outgoing event-propagating associations a from [currentTarget] are evaluated. If *current* contains less instances of a than the impact range of a (i.e., at least one more instance of a may be added to the path description), the current path is extended with a and added to the result. Recursively, the algorithm is then called with this extended path. Initially, the algorithm is called with the following parameters to determine all event-propagating path descriptions for an overlay $\mathcal{O}_i = ([\mathsf{sourceClass}], A_{implicitSub}, A_{eventProp}$:

$$P \leftarrow GetEventPropPathDescriptions(\mathcal{O}_i, [\mathsf{sourceClass}], \emptyset, ())$$

Algorithm 6.4: GetEventPropPathDescriptions

Input: Overlay $\mathcal{O}_i = ([\mathsf{sourceClass}], A_{implicitSub}, A_{eventProp})$
Input: Subscribable [currentTarget]
Input: Recursively found event-propagating path descriptions P
Input: Current event-propagating path description $current = (c_1, ..., c_m)$
Output: Set of valid event-propagating path descriptions P'

1 **foreach** $a \in A_{eventProp}$ *with* $a = [\mathit{currentTarget}].[...] = [\mathit{target}]$ **do**
2 $\quad r \leftarrow$ impact range of a;
3 $\quad$ **if** $f_{ac}(a, current) < r$ **then**
4 $\quad\quad P' \leftarrow P \cup \{current * a\}$;
5 $\quad\quad P' \leftarrow P \cup GetEventPropPathDescriptions(\mathcal{O}_i, [target], P', current * a)$;
6 $\quad$ **end**
7 **end**
8 $P' \leftarrow P' \cup \{([\mathsf{sourceClass}])\}$;

With this algorithm, similar to depth-first search, all paths in the graph of event-propagating associations starting from the overlay's source class and following the rules of impact range, can be determined.

To complete the computation of the triggers, the algorithms for the generation of implicit and explicit triggers still have to be defined. These algorithms are not part of the generic generation procedure, but have to be realized during the implementation of our approach, since they depend on the software architecture of the information system that has to be enhanced with the event-handling functionality.

Thus, instead of describing the implementation details of the two generation algorithms, we only present their interface, consisting of the algorithm's signature and the contracts they have to fulfil.

Algorithm 6.5: CreateExplicitTriggers

Input: Set of path descriptions $P = \{p_1, \ldots, p_n\}$
Result: Set of triggers $T = (t_1, \ldots, t_m)$

1 // algorithm is realized depending on the specific information system
2 // and must fulfil the contract from definition 6.2.2

The interface of the algorithm for the explicit trigger generation is shown in alg. 6.5. A correct implementation is defined as follows:[2]

Definition 6.2.2 (Correct Implementation of CreateExplicitTriggers) Let

$$P = \{p_1, \ldots, p_n\}$$

be a set of path descriptions, each of them starting from the same class [A], i.e.

$$p_i = (\mathsf{[A]}.[\ldots]{:}[\ldots], a_{i2}, \ldots, a_{ix}).$$

Let g be the graph representation of an arbitrary information system instance. Let PI be the set of all path instances in g. Let $Sub_{explicit}$ be a set of explicit subscriptions stored within this information system instance. Let $CreateExplicitTriggers$ be an implementation of algorithm 6.5, so that $CreateExplicitTriggers(P) = \{t_1, \ldots, t_m\}$.

$CreateExplicitTriggers$ is a *correct* implementation of algorithm 6.5, if the following conditions hold:

1. every computed trigger watches modifications of [A], i.e.

$$\forall i \in [1..m] : t_i = (\mathsf{[A]}, Q_i)$$

2. for every input path description starting from source class [A],

$$p_i = (\mathsf{[A]}.[\ldots] = [\ldots], a_{i2}, \ldots, a_{ix}),$$

   there exists at least one trigger containing a query $q \in Q_i$ that determines all explicit subscribers of all targets of path instances matching p_i:

$$\forall p \in P : \forall pi = (pi_1, \ldots, [\ldots]{:}[\ldots].[\ldots] = \mathsf{[targetObject]}{:}[\ldots]) \in PI, pi \subseteq_{matches} p :$$

$$\forall exp \in Sub_{explicit}, exp = (\mathsf{[targetObject]}{:}[\ldots], \mathsf{[subscriberObject]}{:}[\ldots]) :$$

$$\exists t \in T, t = (\mathsf{[A]}, Q) :$$

$$\exists q \in Q : \mathsf{[subscriberObject]} \in eval(q)$$

[2]For better readability, we omit class names, object names and association names that are irrelevant and can have any value and simply write [...] in those cases.

Informally spoken, the definition of a *correct* implementation of the algorithm to create explicit triggers means that the algorithm has to create trigger code which

- detects all relevant updates of an overlay's source subscribable
- and determines all relevant explicit subscribers by correctly following all paths that match the explicit path descriptions.

In a similar manner, we define the contract of a correct implementation of algorithm 6.6 for the generation of implicit triggers has to fulfil:

Algorithm 6.6: CreateImplicitTriggers

Input: Set of path descriptions $P = \{p_1, \ldots, p_n\}$
Result: Set of triggers $T = (t_1, \ldots, t_m)$

```
1 // algorithm is realized depending on the specific information system
2 // and must fulfil the contract from definition 6.2.3
```

Definition 6.2.3 (Correct Implementation of CreateImplicitTriggers) Let

$$P = \{p_1, \ldots, p_n\}$$

be a set of path descriptions each of them starting from class [A], i.e.

$$p_i = ([\mathsf{A}].[\ldots]{:}[\ldots], a_{i2}, \ldots, a_{ix}).$$

Let g be the graph representation of an actual information system instance. Let PI be the set of all path instances in g. Let $CreateImplicitTriggers$ be an implementation of algorithm 6.6, so that $CreateImplicitTriggers(P) = \{t_1, \ldots, t_m\}$.

$CreateImplicitTriggers$ is a *correct* implementation of algorithm 6.6, if the following conditions hold:

1. every computed trigger watches modifications of [A], i.e.

$$\forall i \in [1..m] : t_i = ([\mathsf{A}], Q_i)$$

2. for every input path description $p_i = ([\mathsf{A}].[\ldots]{:}[\ldots], a_{i2}, \ldots, a_{ix})$, there exists at least one trigger containing a query that determines all targets of path instances matching p_i:

$$\forall p \in P : \forall pi = (pi_1, \ldots, [\ldots]{:}[\ldots].[\ldots] = [\mathsf{targetObject}]{:}[\ldots]) \in PI, pi \subseteq_{matches} p :$$
$$\exists t \in T, t = ([\mathsf{A}], Q) :$$
$$\exists q \in Q : [\mathsf{targetObject}] \in eval(q)$$

Finally, two generated triggers have to form a *correct trigger combination* in order to deliver correct results:

Definition 6.2.4 (Correct Trigger Combination) Let

$$t = ([\mathsf{sourceClassT}], Q_t), u = ([\mathsf{sourceClassU}], Q_u)$$

be two triggers as defined in def. 6.2.1. t and u are said to form a *correct trigger combination*, if

$$eval(Q_t) \cup eval(Q_u) = eval(Q_t \cup Q_u)$$

We will present correct implementations of both trigger generation algorithms, using active database technology, in part III.

6.3 Sample Generation Process

Before proving the correctness of the described generation algorithm with respect to the semantic specification from chapter 5, we will illustrate this generation process giving an example.

Our example consists of one overlay: $\mathcal{O} = \{\mathcal{O}_1 = (\mathsf{Lectures}, A_{implicitSub}, A_{eventProp})\}$. The graphic representation of $\mathcal{O}_1$ is shown in figure 6.2.

Algorithm 6.4 computes all valid path descriptions along event-propagating associations in the overlay starting from Lectures using depth-first search and returns[3]

$P =$
$\{(\mathsf{Lectures.takesPlaceIn{:}Rooms}, \mathsf{Rooms.isPartOf{:}Rooms}, \mathsf{Rooms.isPartOf{:}Rooms}),$
$(\mathsf{Lectures.takesPlaceIn{:}Rooms}, \mathsf{Rooms.isPartOf{:}Rooms}),$
$(\mathsf{Lectures.takesPlaceIn{:}Rooms}),$
$(\mathsf{Lectures})\}.$

[3]We omit the tags ≪Subscribable≫ and ≪eventProp≫ for better readability.

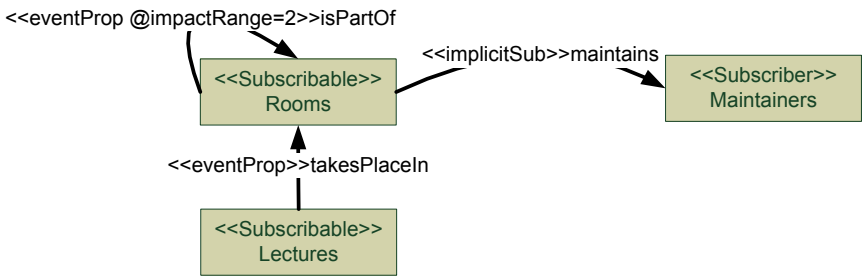


Figure 6.2: Sample overlay

Due to the specification of algorithm 6.2, this is also the result of the explicit path description computation.

P is also the input for algorithm 6.3, computing the implicit path descriptions for overlay $\mathcal{O}_1$. The algorithm iterates over all event-propagating path descriptions (ll. 2-9) and finds all implicit subscriptions in $\mathcal{O}_i$ starting with the target of the path's last association (ll. 3-8).

The following implicit subscriptions are found:

For the path description

(Lectures.takesPlaceIn:Rooms, Rooms.isPartOf:Rooms,Rooms.isPartOf:Rooms),

the implict subscription

Room2.maintains:Maintainer

is found, so the composite path description

(Lectures.takesPlaceIn:Rooms, Rooms.isPartOf:Rooms, Rooms.isPartOf:Rooms, Rooms.maintains:Maintainer)

is added to the result. The remaining path descriptions in P are handled similarly, leading to the overall result

{(Lectures.takesPlaceIn:Rooms, Rooms.isPartOf:Rooms, Rooms.isPartOf:Rooms, Rooms.maintains:Maintainer),
(Lectures.takesPlaceIn:Rooms, Rooms.isPartOf:Rooms, Rooms.maintains:Maintainer),
(Lectures.takesPlaceIn:Rooms, Rooms.maintains:Maintainer)}.

According to algorithm 6.1, these results are then passed to the generation of triggers, which is implementation-specific and can thus not be applied to our example here but will be presented for the prototypic active database implementation in part III.

6.4 Qualitative Analysis

In the following, we will present a qualitative analysis of the generic generation algorithm: we will show that it generates triggers that are correct with respect to the formal concept and that the generated triggers will not cause any cascading update events.

6.4.1 Correctness

We start with the proof that algorithm 6.4 correctly computes all valid event-propagating path descriptions.

Proposition 6.4.1 Let $\mathcal{O}_i = ([\mathsf{A}], A_{implicitSub}, A_{eventProp})$ be an overlay. Algorithm 6.4 computes all valid event-propagating path descriptions as defined in def. 5.5.3.

Proof By construction (the recursive path determination in algorithm 6.4 follows only event-propagating associations), all found paths are event-propagating path descriptions according to definition 5.5.1. Furthermore, the algorithm inserts at most r instances of an association with impact range r, so the returned path descriptions are valid, according to definition 5.5.3. Finally, since algorithm 6.4 is a variant of depth-first search, it is obvious that the algorithm terminates and finds *all* valid event-propagating path descriptions.

Based on this result, we can show that the generated triggers for implicit and explicit subscribers are correct for one overlay.

Proposition 6.4.2 Let $\mathcal{O}_i$ be an arbitrary overlay, let $CreateExplicitTriggers$ be an implementation of algorithm 6.5 that generates explicit triggers that are correct according to definition 6.2.2. For a single-valued set of overlays $\mathcal{O} = \{\mathcal{O}_i\}$, algorithm 6.1 creates triggers that determine explicit subscribers correctly, as defined in def. 5.5.5.

Proof Since the correctness of the computation of the valid event-propagating path descriptions has been shown in proposition 6.4.1, $CreateExplicitTriggers$ receives all valid event-propagating path descriptions in $\mathcal{O}_i$ as input. In addition, we know from definition 6.2.2 (2) that the trigger determines all explicit subscribers of the event-propagating path description targets. Thus, requirements (1) and (3) from definition 5.5.5 are fulfilled.

Due to the depth-first search starting from the overlay's monitored subscribable in algorithm 6.4, every path description starts with the overlay's monitored subscribable and due to the fact that the query of any generated trigger determines only paths matching the input path descriptions (fulfilled requirement (2) in 6.2.2), requirement (2) from definition 5.5.5 is satisfied, too.

Since all three requirements for the correct determination of explicit subscribers are satisfied and requirement (1) in def. 6.2.2 asserts that modifications of the overlay's source subscribable are detected, the postulated claim in proposition 6.4.2 holds.

Proposition 6.4.3 Let $\mathcal{O}_i$ be an arbitrary overlay, $CreateImplicitTriggers$ an implementation of algorithm 6.6 that generates implicit triggers and is correct, as defined in def. 6.2.3. For a single-valued set of overlays $\mathcal{O} = \{\mathcal{O}_i\}$, algorithm 6.1 creates triggers that determine implicit subscribers correctly, as defined in def. 5.5.4.

Proof The correctness of the computation of the valid event-propagating path descriptions has been shown in proposition 6.4.1. Further, algorithm 6.3 computes all path descriptions starting with a valid event-propagating path description and extended by adjacent implicit subscriptions, so the input to $CreateImplicitTriggers$ satisfies the requirements for the path descriptions in item (1) and (3) of definition 5.5.4. Due to requirement (2) in definition 6.2.3, we know that all instance objects that can be reached from the overlay's source subscribable along path descriptions starting with a valid event-propagating path description and ending with an implicit subscription are determined by one of the generated triggers, so requirements (1) and (3) of definition 5.5.4 are completely satisfied.

Similarly to the proof of proposition 6.4.2, requirement (2) is satisfied, too.

As all three requirements for the correct determination of implicit subscribers are satisfied and requirement (1) in def. 6.2.3 asserts that modifications of the overlay's source subscribable are detected, the postulated claim in proposition 6.4.3 holds.

Proposition 6.4.4 (Overall Correctness of Algorithm 6.1) Let $CreateExplicit\text{-}Triggers$ be a correct (def. 6.2.2) implementation of algorithm 6.5, $CreateImplicit\text{-}Triggers$ a correct (def. 6.2.3) implementation of algorithm 6.6.

If both algorithms create correct trigger combinations (cf. def. 6.2.4), algorithm $GenerateTrigger$ (6.1) creates triggers that correctly determine all subscribers for any arbitrary update, according to definition 5.5.6.

Proof Due to propositions 6.4.2 and 6.4.3, both algorithms create correct triggers for one particular overlay.

Let [subscribableObject] be an arbitrary updated object, t be the result of $CreateExplicitTriggers$ and v be the result of $CreateImplicitTriggers$, $\mathcal{O}_i$ be an overlay. Due to algorithm 6.1, a set consisting of both triggers $\{t, v\}$ is returned for $\mathcal{O}_i$. If both algorithms create correct trigger combinations (cf. def. 6.2.4), their results are united, so

$$eval(\{t, v\}) = eval(t) \cup eval(v) = C^{explicit}_{Subscriber}(\mathcal{O}_i) \cup C^{implicit}_{Subscriber}(\mathcal{O}_i) = C^{all}_{Subscriber}(\mathcal{O}_i).$$

Thus, for one overlay $\mathcal{O}_i$, the generated triggers determine the correct subscribers.

Since the outer loop (ll. 2-7) in algorithm 6.1 collects the triggers for all overlays, and as the triggers are correct combinations (def. 6.2.4), we further can derive that, if the algorithm returns a set of triggers $T = \{t_1, v_1, \ldots, t_n, v_n\}$ for an input $\mathcal{O} = \{\mathcal{O}_1, \ldots, \mathcal{O}_n\}$,

$$eval(T) = \cup_{i=1..n}(eval(t_i) \cup eval(v_i)) = C^{explicit}_{Subscriber}(\mathcal{O}) \cup C^{implicit}_{Subscriber}(\mathcal{O}) = C^{all}_{Subscriber}(\mathcal{O}).$$

Thus, proposition 6.4.4 holds.

6.4.2 Avoidance of Event Cascades

A phenomenon that is often observed when using triggers are *cascading triggers*, i.e. triggers updating entities which are also monitored by triggers themselves, thus firing the consecutive trigger, and so on. This is an unwanted effect: in the worst case, endless cascades could occur. These problems can be avoided if the used triggers do not cause cascades at all. We can show that our generation algorithm creates only harmless triggers without cascades:

Proposition 6.4.5 As long as there are no other triggers within the system instance except those that are created by our approach, triggers generated as described above do not cause cascades.

Proof To cause a cascade, one trigger has to modify data of an other monitored class (or by itself). We know that the algorithm generates triggers only for the overlay's source entities tagged as *Subscribable*. By construction, every trigger only determines subscribers to notify and does not modify any data at all. Thus, cascades are impossible.

6.4.3 Discussion on Cycles

Due to our concept and because of the fact that the valid event-propagating path descriptions are computed using depth first search, which by definition does not create endless cycles, infinite cyclic paths can not occur. However, unnecessary "ping-pong" path descriptions can be the result of our generic trigger generation algorithm.

To illustrate this, let us take a look at the excerpt of an overlay, depicted in figure 6.3: this fragment expresses that rooms, which are neighbour to each other, are automatically considered as updated if one of their neighbours (within a distance of four hops) is updated.

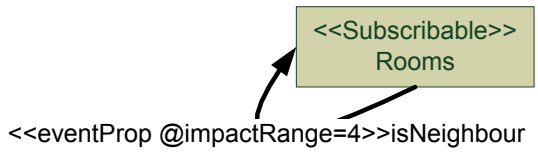


Figure 6.3: Sample overlay leading to ping-pong updates

Due to the reflexive nature of the assocation isNeighbour, for every room A which is neighbour to B, B is neighbour of A, too. Thus, in conjunction with the impact range of 4, event-propagating paths like

(A:Rooms.isNeighbour=B:Rooms, B:Rooms.isNeighbour=A:Rooms,

A:Rooms.isNeighbour=B:Rooms, B:Rooms.isNeighbour=A:Rooms)

are returned as a result by the above-mentioned algorithms.

It is obvious that these "extra" paths are not harmful to the overall correctness of the generated triggers; however, they draw additional and unnecessary performance. It

thus remains an open end of our work, as shown in chapter 12, to avoid such ping-pong paths in order to further streamline our implementation and improve the overall performance.

6.5 Summary

In this chapter, we presented the algorithms to generate triggers that can be used by the event-handling component to automatically determine all relevant subscribers for an arbitrary update. Besides their explanation, we also showed that they generate triggers that are *correct* with respect to the formal event specification and that have no unwanted side effects like trigger cascades or cycles.

Besides correctness, performance is another important issue: in the next chapter, we will go into detail on how to optimize the subscriber-finding queries with respect to their runtime performance, before we will present a prototypic implementation of the above algorithms in part III of this dissertation.

"Have no fear of perfection - you'll never reach it."

Salvador Dalí

7

Generic Optimization Strategy

The following section shows in detail how the performance of the notification facility (generated as shown in chapter 6) can be improved. The idea is to find an optimal indexing strategy, i.e., an optimal mix between the precomputation of event-propagating path instances and their online computation. To this end, implementation-independent cost models will be introduced and used as a measure to determine the quality of an indexing strategy. Examining a small example in different scenarios both theoretically and empirically, we will show that different usage scenarios require different indexing strategies. The chapter concludes with a summary of the results obtained while examining the optimization possibilities.

7.1 Optimization Idea

The most time-consuming part of the event-handling functionality is the computation of path instances matching the event-propagating path descriptions. Based on an overlay, all elements of the information system instance that match one of the event-propagating path descriptions have to be computed at runtime. The longer the path descriptions are, the more expensive the determination of the respective path

instances is. To optimize the performance of an implementation of our event-handling functionality, we have to take a look at this part of our solution.

7.1.1 Computation of Matching Paths

Figure 7.1 shows an example of arbitrary path descriptions. Since several path descriptions from Subscribable to the subscribers Subscriber1 and Subscriber2 are contained in this example, the computation of all paths that conform to one of these path descriptions is very time consuming.

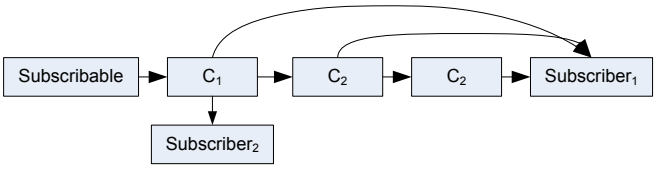


Figure 7.1: Sample path descriptions

To compute all path instances that match the path description, several joins are necessary. In the following, we will talk about *joins*, although our approach is independent of relational database technology. However, no matter which technology the realization of the event-handling component is based on, the underlying information system has to execute queries. For the path description from Subscribable to Subscriber1 in figure 7.1, the following query has to be evaluated:

$$\sigma_{\mathsf{Subscribable} = \mathsf{updatedSubscribable}}(\mathsf{Subscribable} \bowtie \mathsf{C1} \bowtie \mathsf{C2} \bowtie \mathsf{C2} \bowtie \mathsf{Subscriber1})$$

In the following, we will not consider the selection, but take a closer look at the path descriptions and the respective join queries that have to be evaluated to determine the matching path instances.

7.1.2 Optimization by Precomputing Matching Paths

By precomputing (fragments of) these path instances, performance can significantly be optimized. As an example, we propose to precompute all paths within the information systems that conform to the path description

$$(\mathsf{C1}.[\ldots]:\mathsf{C2}),$$

as well as to

$$(\mathsf{C1}.[\ldots]:\mathsf{C2},\ \mathsf{C2}.[\ldots]:\mathsf{C2}]).$$

These precomputed results are stored in what we call *event propagation indices* epI_1 and epI_2, as figure 7.2 illustrates.

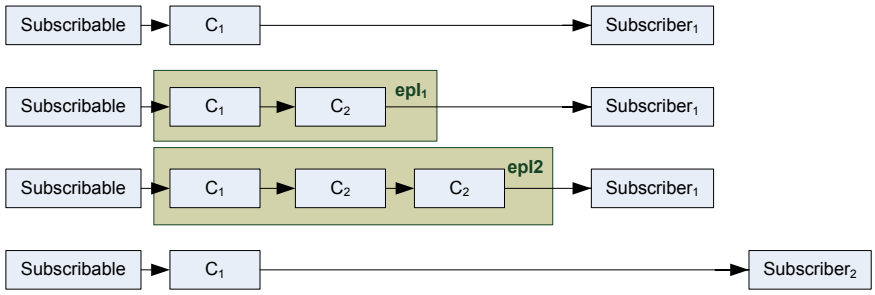


Figure 7.2: Event propagation indices

Using these indices, the runtime system does for instance not have to compute joins like

$$\mathsf{Subscribable} \bowtie \mathsf{C1} \bowtie \mathsf{C2} \bowtie \mathsf{Subscriber1},$$

but instead can lookup epI_1 and compute

$$\mathsf{Subscribable} \bowtie epI_1 \bowtie \mathsf{Subscriber1}.$$

Indices realize a typical trade-off, sacrificing efficient and small-footprint storage for the sake of faster data access. In the following, we will not consider the space consumption of indices; we simply assume that they can be stored in an ideal storage area without space limitations.

As indices have to be maintained as well, their usage offers advantages only if they are used to speed up lookups for paths that seldomly change. In the following, we will describe how to find an optimal strategy for the usage of event propagation indices.

7.2 Strategy for the Usage of Event Propagation Indices

Given an arbitrary path description p, consisting of a combination of associations a_i and event propagation indices ep_i within an overlay, we identify four different scenarios in case of an update:

- P_1 - the update does not affect p at all
- P_2 - the update activates the event-handling mechanism
- P_3 - the update does not activate the event-handling mechanism, but causes the need to update one of the indices ep_i
- P_4 - the update activates the event-handling mechanism *and* induces the need to update one of the indices ep_i

Each of these events causes different costs; joins along associations and/or indices have to be computed, indices have to be queried and indices have to be maintained. Given the likelihood of each of the events P_1 to P_4 and the costs these events cause, the expected value of costs for an arbitrary path description and an arbitrary usage of indices within this path description can be computed. By summing up the expected values for every path within one overlay $\mathcal{O}$, the overall expected costs (with the chosen usage of indices) $E(\mathcal{O})$ can be computed. To find the ideal usage of indices, we have to find a combination of event propagation indices that returns the minimal overall expected costs $E(\mathcal{O})$.

We use the following definitions to explain the usage of event propagation indices:

Definition 7.2.1 (Event Propagation Indices) Let

$$p = (a_1, ..., a_n), a_i \in A_{eventProp}$$

be part of an event-propagating path description. An event propagation index conflating the associations a_1 to a_n into one index is denoted by $epi_{(a_1,...,a_n)}$. The set of all possible event propagation indices is denoted by EPI.

Definition 7.2.2 (Indexed Path Elements) An indexed path element

$$pe \in EPI \cup A_{eventProp}$$

represents either an event-propagating association or an event propagation index. We further introduce the function

$$f_{epiElements} : EPI \cup A_{eventProp} \rightarrow \{(a_1, ..., a_n) | a_i \in A_{eventProp}\},$$

returning the associations that are indexed by epi_i or the association itself, i.e.

$$f_{epiElements}(epi_{(a_1,...,a_n)}) := (a_1, ..., a_n)$$

and

$$f_{epiElements}(a_i) := (a_i).$$

These indexed path elements can then be used within indexed path descriptions, which are defined as follows:

Definition 7.2.3 (Indexed Path Descriptions) Let

$$p = (a_1, ..., a_n), a_i \in A_{eventProp}$$

be an event-propagating path description. If parts of this path description are replaced by one or more event propagation indices consisting of indexed path elements $pe_1, \ldots, pe_m$ as defined in def. 7.2.2, this is denoted by an indexed path description $ip = (pe_1, ..., pe_m)$.

An indexed path description $ip = (pe_1, ...pe_m)$is said to be a *valid representation* of an event propagating path description $p = (a_1, ..., a_n)$ if

$$f_{epiElements}(pe_1) * ... * f_{epiElements}(pe_m) = (a_1, ..., a_n)$$

with $*$ being the concatenation of path description fragments.

7.3 Cost Model

To compute the overall expected costs, a cost model is required. Since our solution is specified independent of any particular implementation technique, we define a generic cost model that predefines the different cost factors that are relevant. For a particular implementation technique like relational databases, XML databases, flat files, etc., the cost model has to be instantiated concretely.

The generic costs can be divided into two groups: costs for the maintenance and access to the event propagation indices and costs that originate from the computation of path instances matching the event-propagating path descriptions have to be evaluated.

7.3.1 Costs of Event Propagation Indices

There are three relevant actions concerning event propagation indices. First, we define the costs for a query to such an index.

Definition 7.3.1 (Costs for *epI*-access) Let EPI be the set of all event propagation indices. The costs that have to be beared whenever an index is queried are denoted by

$$c_{epiAccess} : EPI \rightarrow \mathbb{N}^+$$

As indices have to be maintained, we need to consider the costs for the (re-)computation of an index, too.

Definition 7.3.2 (Costs for *epI*-computation) Let EPI be the set of all event propagation indices. The costs that rise whenever an index is (re-)computed are denoted by

$$c_{epiComp} : EPI \rightarrow \mathbb{N}^+$$

Finally, during maintainance, a recomputed index also has to be stored, which causes additional costs:

Definition 7.3.3 (Costs for *epI*-storage) Let EPI be the set of all event propagating indeces. The costs arising whenever an index is stored within the index store is denoted by

$$c_{epiStore} : EPI \rightarrow \mathbb{N}^+$$

7.3.2 Costs of Paths

To quantify the costs that are needed to compute the transitively updated subscribables along an indexed path description ip (which obvioulsy depends on the different associations and indices within this path), we define the following cost function:

Definition 7.3.4 (Costs for Computation of Matching Path Instances) Let $\mathcal{IP}$ be the set of all indexed path descriptions. The costs that are required to compute all paths within the information system that match a path description $ip \in \mathcal{IP}$ are denoted by

$$c_{path} : \mathcal{IP} \rightarrow \mathbb{N}^+$$

These four cost functions are contained in a cost model, defined as follows:

Definition 7.3.5 (Cost Model) Let $c_{epiAccess}$, $c_{epiComp}$, $c_{epiStore}$, c_{path} be cost functions as defined in definitions 7.3.1, 7.3.2, 7.3.3 and 7.3.4.

These functions constitute a *cost model*, denoted by

$$\mathcal{M}^c = (c_{epiAccess}, c_{epiComp}, c_{epiStore}, c_{path})$$

7.4 Probabilities

In addition to the costs of particular operations, the probabilites for the different events that cause those operations have to be known. We will take a closer look at three different probability models that can be used, shown in figure 7.3. In the following, we will describe the abstract ProbabilityModel, i.e. the different generic probabilities that have to be computed for every information system (in one of the ways shown in figure 7.3).

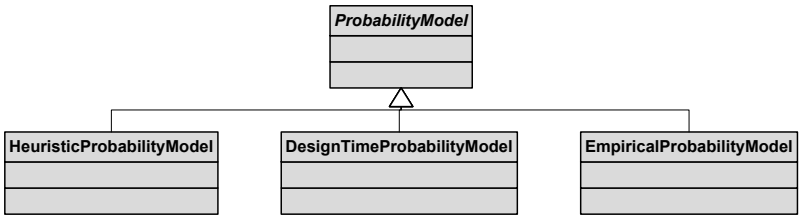


Figure 7.3: Hierarchy of probability models

If we take a look at cases P_1 to P_4 from section 7.2, we identify the following two likelihood functions.

Definition 7.4.1 (Probability for Notification Triggering) For every overlay $\mathcal{O}$, we denote the likelihood that an arbitrary update within the information system requires that the path instances matching the path descriptions within this overlay have to be evaluated (because of an update) by

$$P_{evalIP} : \mathcal{O} \rightarrow [0..1]$$

We also define the likelihood of an update within an arbitrary event propagation index, causing the need to recompute the index:

Definition 7.4.2 (Probability for Index Updates) Let $epi \in EPI$ be an event propagation index. The probability that an arbitrary update within the information system requires that the index is recomputed is denoted by

$$P_{recompEpi} : EPI \rightarrow [0..1]$$

For any instantiation of the generic likelihoods P_{evalIP} and $P_{recompEpi}$, we assume that they are stochastically independent of each other, so that combined events (e.g. two indices being touched by the same update transaction) can easily be handled.

Mapped to the scenarios from section 7.2, we can compute the likelihoods for these four situations within an overlay $\mathcal{O}$ containing the indexed path descriptions $ip_1, ..., ip_n$ as follows (scenario P_1 is not relevant):

The probability $P(P_2)$, i.e. that an overlay $\mathcal{O}$ has to be interpreted because of an update of its source subscribable, has been defined as

$$P(P_2) = P_{evalIP}(\mathcal{O}).$$

Further, the probability that at least one event propagation index has to be recomputed can be computed using the counter-event: the probability that an index ip_i needs *no* update is $1 - P_{recompEpi}(ip_i)$, so the likelihood that none of the indices ip_1 to ip_n needs an update can be computed as

$$\prod_{i=1}^{n}(1 - P_{recompEpi}(ip_i)).$$

Thus, the probability that at least one index has to be updated is

$$P(P_3) = 1 - \prod_{i=1}^{n}(1 - P_{recompEpi}(ip_i)).$$

Finally, the probability of P_4 can be computed as

$$P(P_4) = 1 - ((1 - P(P_2)) \cdot (1 - P(P_3))).$$

These probabilities are represented in a *probability model*, defined as follows:

Definition 7.4.3 (Probability Model) Let P_{evalIP} and $P_{recompEpi}$ be probability functions as defined in definitions 7.4.1 and 7.4.2.

These two function constitute a *probability model*, denoted by

$$\mathcal{M}^P = (P_{evalIP}, P_{recompEpi})$$

7.5 Expected Costs of Path Descriptions

The quality of an indexed path description is evaluated using the expected value for the overall costs of an update. Based on the previously presented generic cost- and probability models, the expected value is computed as follows:

Definition 7.5.1 (Expected Value for Arbitrary Update Costs) Let

$$\mathcal{M}^c = (c_{epiAccess}, c_{epiComp}, c_{epiStore}, c_{path})$$

be an arbitrary cost model as introduced in def. 7.3.5, let

$$\mathcal{M}^P = (P_{evalIP}, P_{recompEpi})$$

be an arbitrary probability model (cf. def. 7.4.3). Let $\mathcal{O}$ be an overlay containing the indexed path descriptions $ip_1, ..., ip_n$ where $ip_i = (pe_{i1}, ..., pe_{im})$. We further denote the set of all event propagation indices epi contained in $ip_1, ..., ip_n$ by EPI.

The expected value for the costs of an arbitrary update is then denoted by

$$E(\mathcal{O})$$

and computed as

$$E(\mathcal{O}) := P_{evalIP}(\mathcal{O}) \cdot \sum_{i=1}^{n} c_{path}(ip_i) + \sum_{epi \in EPI} P_{recompEpi}(epi) \cdot (c_{epiComp}(epi) + c_{epiStore}(epi))$$

Based on this expected value, the quality of the usage of event propagation indices can be evaluated.

7.6 Probability Models

The three concrete implementations of the abstract model shown in figure 7.3 are explained in the following.

7.6.1 Heuristic Probability Model

A very generic way of expressing likelihoods is the usage of heuristics. By assuming equally distributed update probabilites for any attribute or association within the information model, the probabilites $P_{recompEpi}$ and P_{evalIP} can be deferred.

In a heuristic model, we assume that updates are equally distributed over all classes that are adjacent to the associations. Additionally, for every class c, updates activating the event-handling mechanism and updates modifying any of the event propagation indices are assumed to both be equally likely. All these events are assumed to be stochastically independent of each other. Thus, for any overlay $\mathcal{O}$ in an information system with a total of n classes[1], we get

$$P^{heur}_{evalIP}(\mathcal{O}) := \frac{1}{2n}$$

For any event propagation index $epi = (a_1, ..., a_k)$, the probability for a recomputation can be computed (using the counter-event 'none of the classes is updated, affecting the index') as

$$P^{heur}_{recompEpi}(epi) = 1 - (1 - \frac{1}{2(k+1)})^{k+1}$$

Although this model delivers very imprecise results (because the updates in real life systems are usually not uniformously distributed), it can be used for systems that are still under design and whose update behaviour is not yet known to developers.

7.6.2 Designtime Probability Model

If - in contrast to the heuristic model - the designer of an information system can predict the update probabilities, he can specify them at design time. In our concept, this can be expressed by discretely specifying the functions P_{evalIP} and $P_{recompEpi}$ (in

[1] Obviously, a path description consisting of a total of m associations contains $n = m + 1$ classes.

section 10.1.2 we will show how this can be done using UML profiles). In this case, we rely on the designer to assert stochastic independence.

Compared to the heuristic model, this approach requires more knowledge at design time, however the results are expected to be more precise since the system designer's knowledge is assumed to be better than statistic considerations.

7.6.3 Empirical Probability Model

The most precise model can be determined by monitoring the information system and recording the update statistics. By logging the individual updates together with the respective update transaction and correctly deriving the individual likelihoods, stochastic independence can easily be guaranteed. In this work, we will not elaborate such an implementation, but keep this task in mind as an open end (cf. chapter 12).

From an overall view, this probability model delivers the most precise results and does not require any designtime knowledge; however, optimizations at designtime are hardly possible because an information system that is already in use is required. Additionally, monitoring the system's update behaviour during runtime can impose significant performance drawbacks.

7.6.4 Comparison of Probability Models

Table 7.1 illustrates the (dis-)advantages of the probability models with respect to four aspects: is the approach usable already at designtime, does it require knowledge of the system designer considering the system behaviour, does it cause any impact on the running system and, finally, how precise can the optimization be performed. Each of these models can be used to optimize the performance of the event-handling mechanism, depending on the prerequisites and requirements of the actual use case.

	Design time	Knowl. required	Perform. impact	Precision
Heuristic Model	✓	-	-	poor
Probability Model	✓	✓	-	average
Empirical Model	-	-	✓	good

Table 7.1: Comparison of probability models

7.7 Cost Models

There are several possibilites of how to instantiate the generic cost functions (shown in figure 7.4), depending on the desired precision and on the chosen realization of the concept: a heuristic cost model, using expected costs for joins based on the classes' cardinalities and the join selectivities, a database-based cost model which uses the internal cost models of relational databases (thus being applicable to relational implementations only) and an empirical cost model, based on observations of the actual event-handling system.

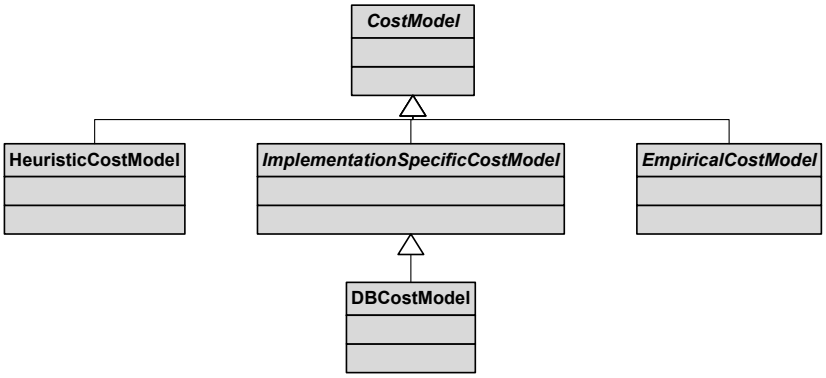


Figure 7.4: Hierarchy of cost models

7.7.1 Heuristic Cost Model

To evaluate the usability and adequacy of particular event propagation indices, heuristic cost models can be used. These cost models can be utilized using only a model of the information system and an estimation about the cardinalities of classes and about the join selectivities between classes. However, the lack of knowledge about the concrete implementation of the event-handling system results in inaccurate and rough evaluations and can thus only be used to quickly purge the search space of possible index assignments, so that fewer alternatives have to be compared to find an optimal solution.[2]

[2] To guarantee that the heuristic purge process does not remove potentially optimal solutions, we would have to prove that the heuristic cost model induces the same order on the set of index assignments as the empricial specific cost model does. We do not prove this, so we can only propose the purge process as a heuristic without guaranteed correctness or a guaranteed limited deviation of the optimum.

Definition 7.7.1 (Cardinality of Classes) Let $c \in C$ be an arbitary class. The cardinality of this class (i.e. the number of instances of this class in the information system) is denoted by

$$|c|.$$

Definition 7.7.2 (Join Selectivity) Let $\mathcal{P}$ be the set of all path descriptions, let

$$p = (a_1, ..., a_n) \in \mathcal{P}$$

be a path description, connecting the classes c_1 to c_{n+1} via the associations a_1 to a_n. The join selectivity

$$sel_{join} : \mathcal{P} \rightarrow \mathbb{N}^+$$

defines the size of the result of the join $c_1 \bowtie ... \bowtie c_{n+1}$ for the path description p.

Since the costs to access an event propagation index are related to the size of the index, which equals the join selectivity of the indexed classes, we further define the following heuristic cost functions:

Definition 7.7.3 (Heuristic costs for *epI*-access) For an arbitrary event propagation index *epi*, we define the heuristic access costs by

$$c^{heur}_{epiAccess}(epi) := sel_{join}(f_{epiElements}(epi))$$

The costs for the computation of event propagation indices are estimated based on the cardinalities of the individual joins. The join operator is associative, so some information systems (like relational databases, e.g.) change the join order to find an optimal reordering to minimize join costs. To consider this re-ordering in our cost model, we introduce the following join reordering function.

Definition 7.7.4 (Join Reordering) Let $ip = (pe_1, ..., pe_n)$ be an indexed path description containing the path elements pe_1 to pe_n. We use the *join reordering* function

$$\phi : \{1, ..., n\} \rightarrow \{1, ..., n\}$$

to express the join order. For a path containing 4 classes, the join reordering function would be interpreted as follows:

For the trivial case of no join order optimization, we use the *left-to-right ordering* ϕ_{lr},

$$\phi_{lr}(n) := n.$$

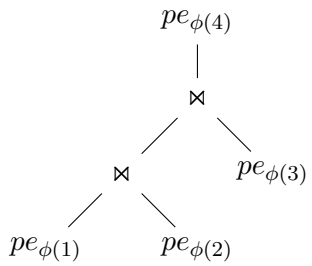


Figure 7.5: Join order expressed by ϕ

The set of all valid join reorderings for an indexed path description ip is denoted by

$$\Phi^{ip}.$$

Definition 7.7.5 (Heuristic Costs for *epI*-computation) Let $epi \in EPI$ be an event propagation index, indexing the path description $p = (a_1, ..., a_m)$, connecting the classes $c_1, ..., c_n$. Let ϕ be a valid join reordering for p. The heuristic computation costs are defined recursively by

$$c^{heur}_{epiComp}(epi, \phi) := c^{heur}_{epiComp}(epi, \phi, n)$$

where

$$c^{heur}_{epiComp}(epi, \phi, n) := c^{heur}_{epiComp}(epi, \phi, n-1) + |c_{\phi(n)}| + c^{heur}_{epiComp}(epi, \phi, n-1) \cdot |c_{\phi(n)}|$$

and

$$c^{heur}_{epiComp}(epi, \phi, 2) := |c_{\phi(1)}| + |c_{\phi(2)}| + |c_{\phi(1)}| \cdot |c_{\phi(2)}|$$

Although most cost models (especially in the area of relational databases) do not consider storage costs, because they can hardly be set in relation to access and computation costs [HR01], we decided to incorporate them because the storage of an index significantly influences the overall runtime behaviour, independent of the concrete implementation. We thus estimate the storage costs by the number of indexed paths that have to be stored, multiplied by a factor k_{rw} approximating the cost relationship between writing and reading access.

Definition 7.7.6 (Heuristic costs for *epI*-storage) Let $epi \in EPI$ be an event propagating index. The costs for storing this index are heuristically defined by

$$c^{heur}_{epiStore} := k_{rw} \cdot c^{heur}_{epiAccess}$$

where

$$k_{rw} > 1$$

The heuristic approximation of the path evaluation costs is again based on the assumption that joins between path elements (i.e. classes or indices) are performed according to the join reordering. Using the index access costs and the join selectivities, we define the following heuristic cost function:

Definition 7.7.7 (Heuristic Costs for Path Computations) Let ip be an indexed path description $ip = (pe_1, \ldots, pe_n)$. Let ϕ^{epi} be a valid join reordering for ip. The heuristic path evaluation costs are defined by

$$c^{heur}_{path}(ip, \phi^{epi}) := c^{heur}_{path}(ip, \phi^{epi}, n)$$

where

$$c^{heur}_{path}(ip, \phi^{epi}, n) :=$$
$$c^{heur}_{path}(ip, \phi^{epi}, n-1) + c^{heur}_{access}(pe_{\phi^{epi}(n)}) + c^{heur}_{path}(ip, \phi^{epi}, n-1) \cdot c^{heur}_{access}(pe_{\phi^{epi}(n)})$$

and

$$c^{heur}_{path}(ip, \phi^{epi}, 2) := c^{heur}_{access}(pe_{\phi^{epi}(1)}) + c^{heur}_{access}(pe_{\phi^{epi}(2)}) + c^{heur}_{access}(pe_{\phi^{epi}(1)}) \cdot c^{heur}_{access}(pe_{\phi^{epi}(2)})$$

Based on these cost functions, we can define an *optimal reordering*:

Definition 7.7.8 (Optimal Join Reordering for Indexed Path Descriptions) Let $ip = (pe_1, ..., pe_n)$ be an indexed path description. Let Φ^{ip} be the set of all valid join reorderings for ip, let $\phi \in \Phi^{ip}$ be one of those reorderings.

ϕ is said to be an *optimal join reordering* (denoted by $\phi_{optimal}$), iff

$$c_{path}(ip, \phi) = min\{c_{path}(ip, \phi') | \phi' \in \Phi^{ip}\}$$

Using this cost model in conjunction with the formula for the overall expected value from section 7.5, different scenarios of index usage can be compared to each other without in-depth knowledge about the platform of the event-handling application. We will present an example in section 7.8.

7.7.2 DBCostModel

In contrast to the heuristic cost model, implementation specific cost models know about the actual costs for the mentioned operations using a particular implementation. This cost model can be instantiated for XML databases, object databases, text files or relational databases using a particular *DBCostModel*.

In case the event-handling system is built upon relational databases, the cost model can be tailored to this use case by using a specific database adapted *DBCostModel*. Since cost models for relational databases have been studied in detail, precise estimations about different database operations can be computed.

7.7.3 Empirical Cost Model

Finally, the empirical cost model is based upon observations of the system in use: every component of the cost model (i.e. costs for updating indices, ...) has to be measured using the real event-handling system.

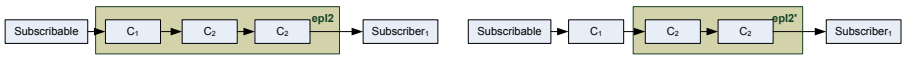
For relational database systems, optimizers can for instance be queried to return the actual database costs for complex queries like the determination of event-propagating paths or for index maintenance.

Obviously, this cost model returns the precisest results; however, it is often impossible or too expensive to conduct measurements within a productive system. Moreover, optimizations usually have to take place *before* an event-handling system goes into production, so that no empirical results are present yet.

On the other hand, such observations can regulary be conducted while the system is in use, so that the overall event-handling system can be auto-tuned by automatically adapting to any changed costs (or to new usage statistics, resulting in different update likelihoods). This aspect is, however, not in the focus of our work, so we will list this task as an open end in chapter 12.

7.8 Sample Comparison of Indexing Alternatives

To illustrate the defintions and to show the relevance of our optimization efforts, we will compare two different ways of indexing path descriptions, based on the heuristic cost model from section 7.7.1. The following example has already been presented as one of the path descriptions in figure 7.2. Two ways of indexing this path description are presented in figure 7.6.



(a) Usage of index: Alternative A (b) Usage of index: Alternative B

Figure 7.6: Alternative usage of indices

In this example, $p = (\mathsf{C1}.[\ldots]{:}\mathsf{C2}, \mathsf{C2}.[\ldots]{:}\mathsf{C2})$ is the event propagating path description between Subscribable and Subscriber1. This path can be represented in two ways: either by

$$ip_A = (epi2)$$

or by

$$ip_B = (\mathsf{C1}.[\ldots]{:}\mathsf{C2}, epi2'),$$

where

$$f_{epiElements}(epi2) = (\mathsf{C1}.[\ldots]{:}\mathsf{C2}, \mathsf{C2}.[\ldots]{:}\mathsf{C2})$$

and

$$f_{epiElements}(epi2') = (\mathsf{C2}.[\ldots]{:}\mathsf{C2}).$$

Besides the two indexing strategies shown above, we also evaluate a third alternative without any event propagation indices at all. We evaluate four different scenarios, differing in the update probability of every class, the cardinality of each class and the respective join selectivities. Each scenario is examined under the premise that the underlying information system either is able to determine an optimal join order ($\phi_{optimal}$), or that it does not optimize the join order at all (ϕ_{lr}). We furthermore assume a write-to-read factor $k_{rw} = 1.5$, i.e. storing a tuple causes 150 percent of the costs of reading it. The expected costs for every scenario are computed using the above-mentioned formula for the expected value.

Independent of the scenario, there are two ways of computing $\mathsf{C1} \bowtie \mathsf{C2} \bowtie \mathsf{C2}$. We thus get join costs

$$c_{join}((\mathsf{C1} \bowtie \mathsf{C2}) \bowtie \mathsf{C2}) = |\mathsf{C1}| \cdot |\mathsf{C2}| + sel_{join}(\mathsf{C1}, \mathsf{C2}) \cdot |\mathsf{C2}|$$

or

$$c_{join}(\mathsf{C1} \bowtie (\mathsf{C2} \bowtie \mathsf{C2})) = |\mathsf{C1}| \cdot sel_{join}(\mathsf{C2}, \mathsf{C2}) + |\mathsf{C2}| \cdot |\mathsf{C2}|,$$

depending on the join order. In the following, the optimal join order was used for every scenario without explicitly presenting the mathematical comparison of both alternatives.

To quickly visualize the different scenarios, we use the graphical representation shown in figure 7.7. The thickness of the borders represent the update probability of a class: a dotted border means low update probability, a normal border represents an average probability and a fat border indicates high update likelihood. The cardinalities of each class are represented by the sign above the class: '+' means high, 'o' means average and '-' represents low cardinality. The same signs are used for the indication of the join selectivity between two classes (drawn below the association) and between all three classes (below the bracket).

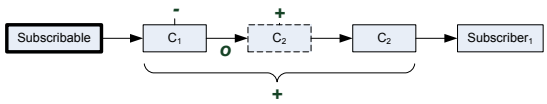


Figure 7.7: Graphical representation of scenarios

Based on these scenarios[3], the expected values are computed and the results are presented in tabular form and explained briefly.

7.8.1 Scenario 1

In the first scenario, the update likelihoods of C1 and C2 are very low compared to the update probability of the subscribable. The cardinalities and join selectivities are given in table 7.2 and visualized in figure 7.8.

Computing the overall expected update costs for all three alternatives, we get the results as listed in table 7.3. As the results show, the indexing alternative B performs

[3]To avoid the influence of the join Subscribable $\bowtie$ C1, the cardinalities of Subscribable and C1 were equally set for every scenario, with every Subscribable being connected to exactly one C1.

Class c	$card$	sel_{join}	P_{update}
C1	5,000		0.01
C2	5,000		0.01
C1 ⋈ C2		5	
C2 ⋈ C2		500	
C1 ⋈ C2 ⋈ C2		20	
Subscribable	5,000		0.98

Table 7.2: Cardinalities and probabilities for figure 7.2, scenario 1

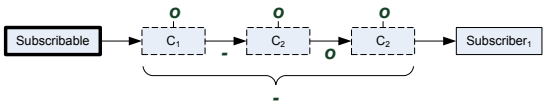


Figure 7.8: Graphical representation of scenario 1

best in case the information system does not support optimal join ordering. If an optimal join strategy is determined by the information system, alternative A using the large index performs best.

	Alternative A	Alternative B	No EPIs
$\phi_{optimal}$	606,224	2,808,417	24,568,609
ϕ_{lr}	606,224	27,215,297	49,049,004

Table 7.3: Expected values for scenario 1

7.8.2 Scenario 2

The second scenario, shown in table 7.4 and figure 7.9, has slightly different probability parameters: class C1 is much more likely to be updated than C2 or Subscribable.

The results presented in table 7.5 show that in this scenario, alternative B is best in both cases. Since C1 is updated frequently, while C2 is updated infrequently, the separate index for C2 ⋈ C2 performs better than the complete index. Finally, due to the fact that Subscribable is not updated very often, the approach using the index performs better than the alternative without indices at all.

Class c	$card$	sel_{join}	P_{update}
C1	50,000		0.30
C2	100		0.05
C1 ⋈ C2		250,000	
C2 ⋈ C2		300	
C1 ⋈ C2 ⋈ C2		2,000	
Subscribable	50,000		0.10

Table 7.4: Cardinalities and probabilities for figure 7.2, scenario 2

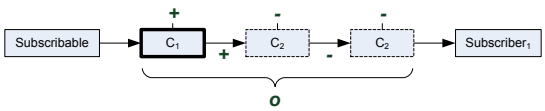


Figure 7.9: Graphical representation of scenario 2

7.8.3 Scenario 3

In the third scenario (table 7.6 and figure 7.10), the update probability for Subscribable is significantly lower than the other update probabilities. In addition, the selectivity of C2 ⋈ C2 is very high, thus minimizing the use of an index for this join.

As a result (cf. table 7.7), both indexing strategies perform worse than the non-indexed approach, even if the information system does not determine the optimal join strategy. This can be explained by two factors: first, the high selectivity of C2 ⋈ C2 and C1 ⋈ C2 ⋈ C2 makes access to both index alternatives almost as expensive as an online computation; second, the low update probability of Subscribable leads to many unnecessary (and, due to the cardinalities of C1 and C2, expensive) updates of the index, because the index is queried seldomly.

	Alternative A	Alternative B	No EPIs
$\phi_{optimal}$	15,503,377	11,510,762	11,511,250
ϕ_{lr}	21,065,868	251,515,562	253,040,020

Table 7.5: Expected values for scenario 2

Class c	$card$	sel_{join}	P_{update}
C1	1,000		0.2
C2	1,000		0.2
C1 ⋈ C2		800	
C2 ⋈ C2		8,500	
C1 ⋈ C2 ⋈ C2		200,000	
Subscribable	1,000		0.02

Table 7.6: Cardinalities and probabilities for figure 7.2, scenario 3

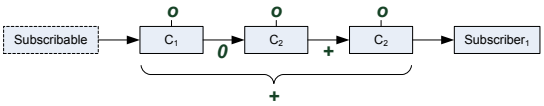


Figure 7.10: Graphical representation of scenario 3

7.8.4 Scenario 4

Scenario 4 (table 7.6 and figure 7.10) differs from the previous scenarios in two aspects: first, the overall selectivity of C1 ⋈ C2 ⋈ C2 is very low. Second, updates of C1 and C2 are much less likely than updates of Subscribable.

The results, presented in table 7.9, yield that alternative B is by far better than any other solution (the significance of the advantage is much higher than in scenario 2), independent of the join order. This is easily explainable, since the index is queried very often (updates of Subscribable are very likely) *and* small enough to offer a significant advantage being queried.

	Alternative A	Alternative B	No EPIs
$\phi_{optimal}$	4,929,692	393,180	52,112
ϕ_{lr}	4,929,692	393,180	56,116

Table 7.7: Expected values for scenario 3

Class c	$card$	sel_{join}	P_{update}
C1	1,000		0.05
C2	1,000		0.05
C1 ⋈ C2		1,000	
C2 ⋈ C2		1,000	
C1 ⋈ C2 ⋈ C2		2,000	
Subscribable	1,000		0.40

Table 7.8: Cardinalities and probabilities for figure 7.2, scenario 4

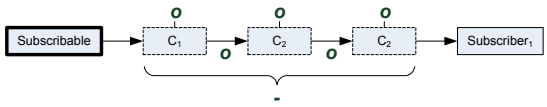


Figure 7.11: Graphical representation of scenario 4

7.8.5 Empirical Validation of the Results

To validate the theoretical results, we implemented the four scenarios using the relational database system Microsoft SQL Server 2005 [Micc]. Since SQL Server does not support materialized views containing self referencing joins (C2 ⋈ C2 in our example), the event propagation indices were realized using regular tables that were filled using database triggers (see appendix B.2). All scenarios were built according to their specification, and every alternative was measured using 1000 updates, distributed among the different classes according to the specified probabilities. As a final result, the average time per update was measured and the average of three measurements was recorded.[4] SQL Server internally optimizes the order of the joins, so the results had to be compared to the respective expected optimal reordering $\phi_{optimal}$.

[4]The scenarios were run on a virtual server under Windows XP, 2.2 GHz Core2Duo with 1.5 GB of RAM.

	Alternative A	Alternative B	No EPIs
$\phi_{optimal}$	1,006,917	851,775	1,202,400
ϕ_{lr}	1,006,917	851,775	1,202,400

Table 7.9: Expected values for scenario 4

To validate the assumed write-to-read factor of 1.5 we ran several tests comparing update costs against the corresponding read costs, which resulted in an actual average write-to-read factor of 1.43, thus confirming our assumptions.

The overall results are presented in table 7.10.

	Expected Costs (cost units / update)			Actual Costs (ms / update)		
Scenario	A	B	C	A	B	C
1	606,224	2,808,417	24,568,609	10.3	23.8	58.9
2	15,503,377	11,510,762	11,511,250	96.2	61.4	68.4
3	4,929,692	393,180	52,112	828.2	25.1	1.9
4	1,006,917	851,775	1,202,400	7.2	6.9	7.3

Table 7.10: Empirical Validation of Expected Costs per Update

These empirical results first and foremost show that the actual costs behave as expected: the theoretically best indexing strategy is actually the ideal solution. However, the differences between the alternatives are not as significant as predicted. This can be explained by the internal use of caching mechanisms and specialized join algorithms within the database system, which lead to better performance whilst not improving the maintenance of the indices. Thus, especially the non-indexed solution mostly performs better than expected.

Altogether, our tests showed clearly that the theoretical cost model that we proposed resembles the real life behaviour of the event-handling system, thus making it a powerful means of evaluating different index stragies with respect to their performance. Therefore, we decided to use this cost model in our prototypic implementation, which will be presented in part III of this dissertation.

7.8.6 Sophisticated Index Maintenance Algorithms

In our approach, the maintenance of the event propagation indices has either been left completely to the underlying platform (if the platform provides any means of view materialization) or been hand-built by re-building the index whenever one of the participating entities is updated.

Obviously, this is the worst possible solution to maintain the indices. To improve maintainance performance, sophisticated index maintenance are necessary, so that we can identify an open task which is presented in section 12 of this dissertation. Thus,

we do not go into detail regarding these improvements but postulate that such modifications can only *improve* the performance of any event propagation index. Thus, our approach of finding an optimal indexing strategy can be used without modifications; we assume that sophisticated maintenance algorithms do not affect the proportions between different strategies, so that the best solution remains best, also when making use of such improvements.

7.8.7 Results

We can summarize the following results:

- As we already expected, due to the different scenarios with different parameters in real-life use cases, we can state that the general use of indexes is not appropriate. Instead, the contextual situation has to be analyzed individually and tailored indexing strategies - which can be determined with our approach - have to be applied.
- According to our model, the relevant parameters to determine the ideal index usage are: cardinalities of the classes, the join selectivities between those classes along the path description and the update likelihoods of all participating classes.
- The proposed cost model is suitable in so far that the expected costs are able to predict the actual runtime behaviour, at least in a relational database model, which is what we verified.
- There are scenarios in which indices should not be used at all, because online computation is more efficient.

Since - as we were able to show when analyzing the use cases - all of these different scenarios are present in practice, and since the parameters also tend to change over time, we can also conclude that an approach to automatically determine the optimal indexing strategy is in fact necessary. In the next section, we will show how to not only evaluate strategies under different scenarios, but how to also find an optimal indexing strategy for a given use case.

7.9 Determining the Optimal Index Usage

With the above-mentioned approach, different indexing strategies can be compared to each other and the best of those strategies can be chosen. To find an optimal strategy for a given (non-indexed) path description p, all possible partitionings of the path description into indices have to be evaluated, i.e. the whole search space has to be evaluated with respect to the cost model.

To approximate the complexity of this search space, we look at the number of valid partitionings of a path description.

Definition 7.9.1 (Number of Partitionings of a Path Description) Let

$$p = (a_1, ..., a_n)$$

be a (non-indexed) path description with $n-1$ associations, i.e. containing a total of n classes. We denote the number of path partitionings into classes and event propagation indices by

$$part : \mathcal{P} \rightarrow \mathbb{N}^+$$

Thesis 7.9.2 Let $p = (a_1, ..., a_n)$ be a (non-indexed) path description with $n - 1$ associations, i.e. containing a total of n classes. The number of valid path partitionings is

$$part(n) = 2^{n-1}$$

Proof The proof is lead inductively: figure 7.12 shows an arbitrary path with $n > 2$ classes.

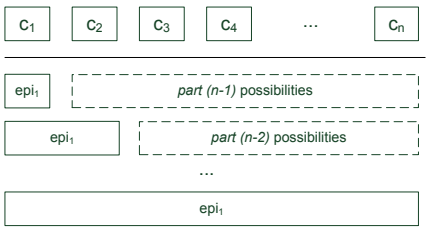


Figure 7.12: Recursive construction of valid path partitionings

Obviously, the first partition (starting with C1) can be of length i, $1 < i < n$. For the remaining classes, $part(n - i)$ possibilities remain. Thus, we get

$$part(n) = part(n - 1) + part(n - 2) + ... + part(2) + part(1) + 1$$

We define $part(0) := 1$, so we get

$$part(n) = \sum_{i=0}^{n-1} part(i)$$

With $part(0) = 1$ (by definition), $part(1) = 1$ and $part(2) = 2$, it remains easy to show that the formula can be simplified to

$$part(n) = 2^{n-1}. \quad \blacksquare$$

For an overlay $\mathcal{O}$ with m paths, each of them with a maximum amount of n classes, a total of

$$O(m \cdot 2^{n-1}) = O(2^n)$$

possibilities in the search space have to be evaluated.

Although this means exponential complexity with respect to the longest path description within an overlay, we decided not to use any specific optimization techniques like dynamic programming for two reasons: first of all, our research showed that the maximal length and the number of path descriptions in one overlay are bounded by design: path descriptions longer than four or five classes do not appear in real-life scenarios, so that the search space remains maintainably small. Second - and most important - the evaluation of the different alternatives is done once, at design time, and thus does not affect runtime behaviour, which means that the duration of this optimization is irrelevant for the target system at all.

7.10 Estimated Behaviour Depending on Path Length

In the following, we will finally analyze the response-time behaviour of the event-handling component depending on the length of an event-propagating path description. The results were determined using the heuristic cost model from section 7.7.1. For this analysis, we differentiate between highly connected and sparsely connected object graphs.

7.10.1 Highly Connected Object Graphs

As an example for a highly connected object graph, we consider the model of a graph containing n classes (with n being the overall path length). Each of those classes is assumed to be of cardinality 100, where each join selectivity between two classes is specified as $10,000$, i.e. each object is connected to each other. Thus, the overall number of paths conforming to the path description is n^{10000}.

Two indexing strategies are compared to each other: no index vs. the use of a total index, i.e. an index that precomputes the whole path description. We analyze two cases for each strategy: in the first case, 10 percent of all accesses to the event-handling system are read accesses, i.e. all relevant subscribers have to be determined. The other 90 percent are update transactions, updating the index (if any) and not leading to a determination of subscribers.

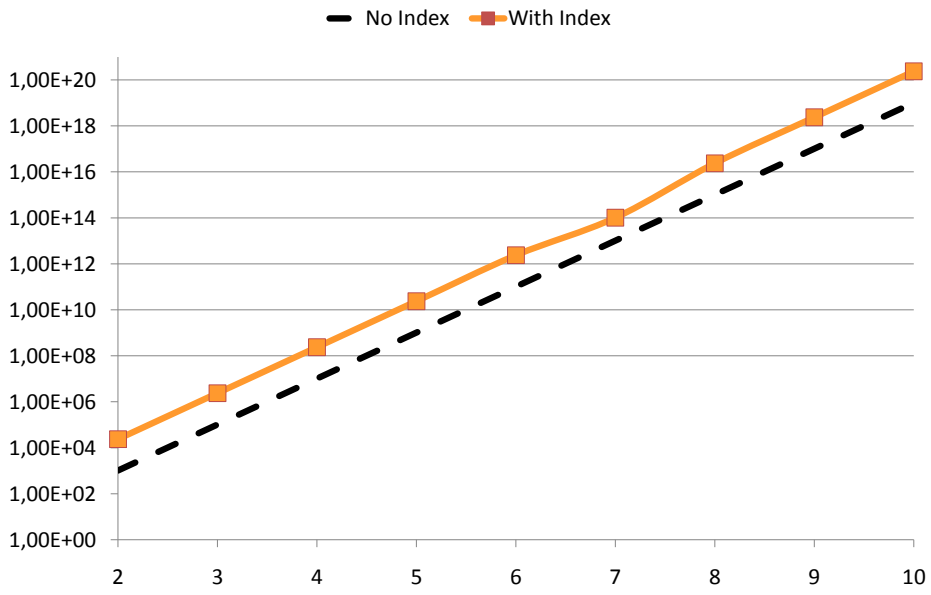


Figure 7.13: Average time per update in highly connected graph, few read and many update transactions (logarithmic scale)

Figure 7.13 shows the estimated average costs per transaction: obviously both alternatives scale exponentially. This is due to the exponential growth of paths conforming to the path description. The non-indexed strategy performs slightly better because the extra costs for the index maintenance are not compensated, since the index is "complete".

In the contrary scenario (90 percent read and 10 percent update transaction), the same behaviour can be observed (figure 7.14). In this case, both alternatives are equally bad because accessing the index is only neglibly faster than computing all relevant paths.

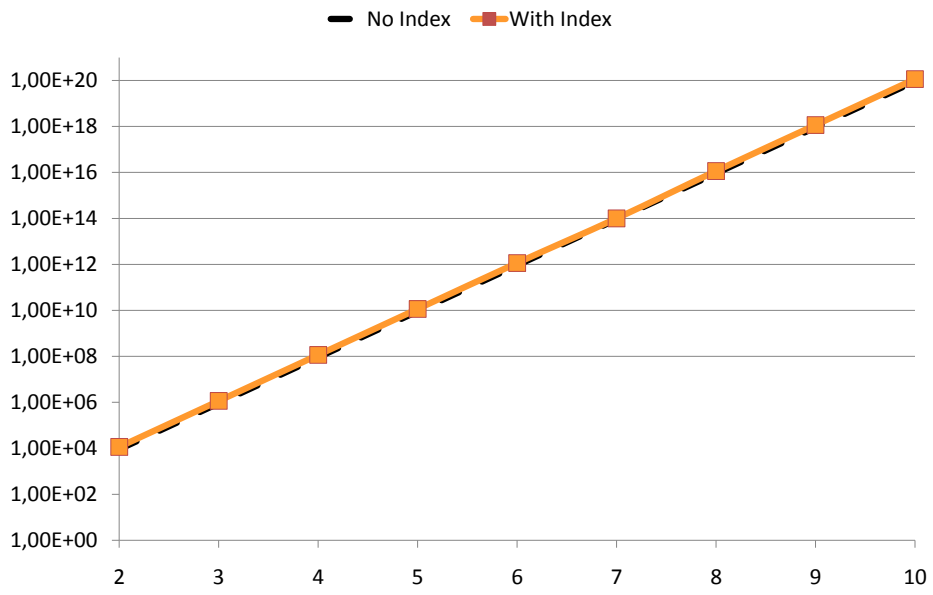


Figure 7.14: Average time per update in highly connected graph, many read and few update transactions (logarithmic scale)

Although the proposed optimizer chooses the solution with minimal expected costs per update for any particular scenario (i.e. for a given n), highly connected graph structures represent the worst situation, because exponential behaviour can not be prevented.

7.10.2 Sparsely Connected Object Graphs

In contrast to highly connected graph structures, the response-time behaviour of a sparsely connected object graph is evaluated: each of the n classes is again assumed to be of cardinality 100, but the join selectivity between two classes is only 1, i.e. there is only one join between two neighboured classes. We further assume that the total join returns only 1 result, i.e. there is only one event-propagating path at all.

Again, we distinguish between two usage statistics. Figure 7.15 shows the expected results for 10% read and 90% update transaction.

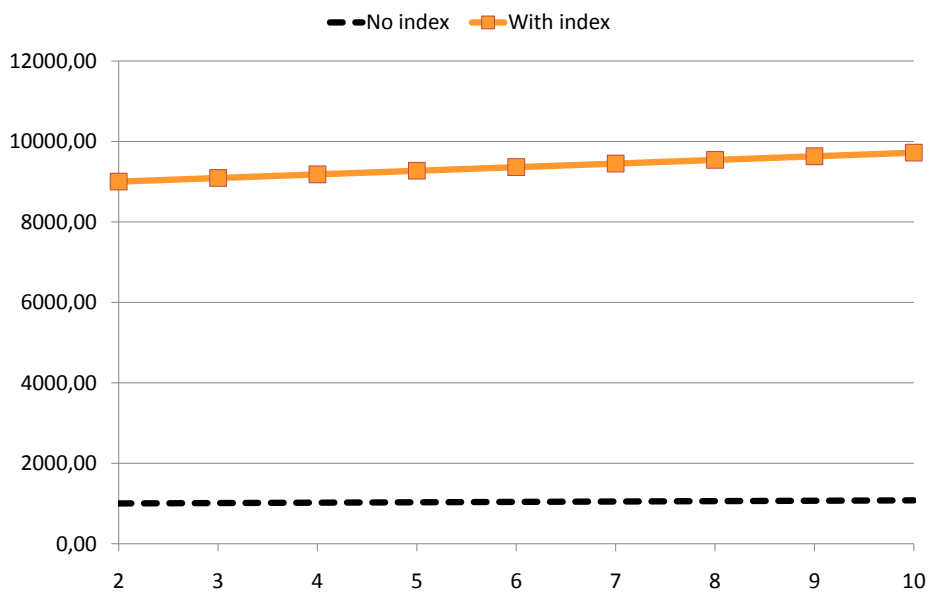


Figure 7.15: Average time per update in sparsely connected graph, few read and many update transactions

This time, both approaches scale linear. Due to the high rate of updates causing an index recomputation, the indexed solution is significantly slower.

A different result is yielded for the contrary case in figure 7.16: this time, an index offers significant advantage because it can be used in 90% of all cases (with only very

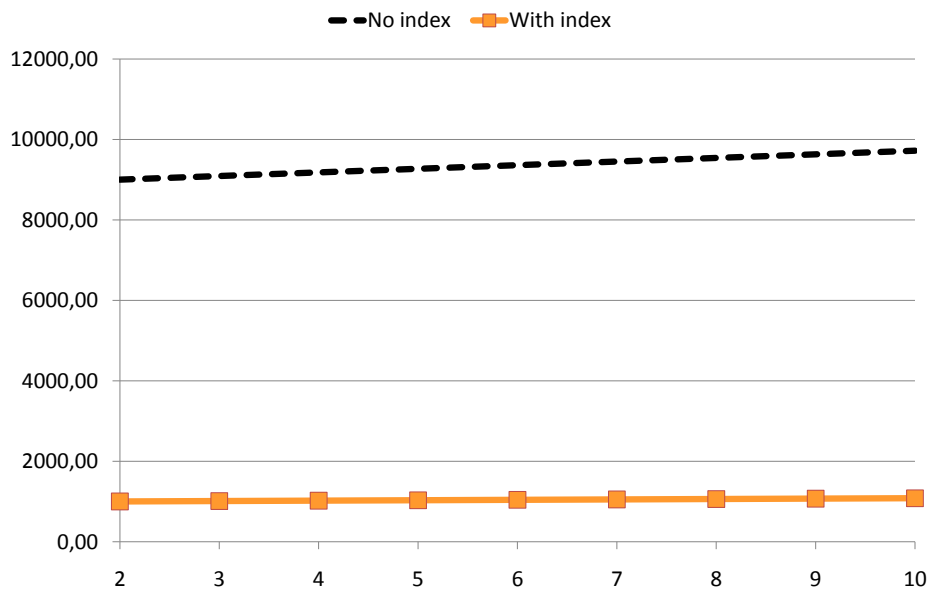


Figure 7.16: Average time per update in sparsely connected graph, many read and few update transactions

low access costs), whilst having to be rebuilt only in 10% of all transactions. However, both solutions again scale linear.

7.10.3 Consequences

As a consequence of these results, we conclude that - in real-life scenarios without fully connected graph structures - the overall scalability is acceptable. Developers applying our approach have to consider that long path descriptions for event propagation heavily impact the overall performance. However, since event-propagating path descriptions are usually of limited length (the real-life scenarios we encountered contained event-propagating path descriptions of a maximum length of three to four only), the overall behaviour is applicable for productive use.

7.11 Summary

In this chapter, we motivated why to use indices for the computation of event-propagating paths. Using a heuristic cost- and probability model, we also proved that under different contexts, i.e. with different parameters like cardinalities, update probabilities and join selectivities, different strategies can be optimal. We were also able to show that there are $O(2^n)$ different ways of how to partition a path description into different indices. Examining this search space in a brute force way, we showed how an optimal strategy can be found.

As a first conclusion, we can argue that the heuristic probability model can be used as a rule-of-thumb to evaluate practical applications, although scenarios with uniformous update likelihoods usually do not appear in real life use cases. For better results, the real (or estimated) update probabilities should be taken into account (either being specified by the system designer or being derived from the actual system). Nevertheless, the heuristic probability model proved to return plausible results when applied to different scenarios and can thus be used for design-time approximations of the later costs. However, this model also requires designers' knowledge concerning the cardinalities and the join selectivities which have to be specified at design time.

From a comprehensive view, we were able to show that the index optimizer is an important fragment of the overall architecture that has to be included in the design process by default and can even - if the overhead for regularily determining the context parameters can be coped with - be used, in conjunction with the runtime lifecycle from chapter 4, to regularily auto-tune the system and adjust it to new usage scenarios.

These results will be applied in the context of relational databases and MDA in part III, where the whole optimization concept will be realized using materialized views.

Part III

The Model Driven Implementation for Active Databases

"For a list of all the ways technology has failed to improve the quality of life, please press three."

Alice Kahn

8 Technology Selection

To start the description of our prototypic implementation of the generic approach presented in the previous part of this dissertation, we are going to give an overview of the technologies we chose. After briefly motivation our selection, we will introduce the reader to the key concepts of Model Driven Architecture, Active Database Technology and Materialized Views, limited to those aspects that are required to understand the remainder of this dissertation. In addition, some traditional fields of application for the presented technologies will be named. Like for all chapters, we end our overview with a short summary.

8.1 Motivation for Technology Selection

The selection of technologies was mainly motivated by our major use case *Stud.IP*, which was already introduced in chapter 2. Since *Stud.IP* stores its data in relational databases, the usage of technologies in the field of databases suggested itself. Whenever it comes to the detection and handling of updates in relational databases, *Active Database Technology* can be considered the most suitable means to develop an adequate solution.

However, this selection only predetermines the target technology for the runtime event-handling component; what remained to choose was a technology for the specification of the event-handling meta-model and of actual event-handling models based on this meta-model, as well as for the development of transformations of event-handling models into runtime components. Both aspects are ideally covered by a technology currently in the focus of both practitioners and researches: *Model Driven Architecture (MDA)*.

Finally, our generic optimization approach had to be implemented, too. A technology that proved suitable for the pre-computation and storage of queries (which is what the event-propagating indices, introduced in the previous chapters, actually do) when working with relational databases are *Materialized Views*, which constitute the third building block of our prototypic implementation.

In the following, we will present the three selected technologies.

8.2 Model Driven Architecture

Our introduction to MDA will be divided into two parts: a description of the model driven architecture paradigm as promoted by the Object Management Group (OMG) and an overview of AndroMDA, an open-source implementation of the MDA paradigm. We will start with a description of MDA in general[1].

8.2.1 MDA in General

MDA [Objb] is a framework for the development of software, defined by the Object Management Group (OMG) [Objc]. Key feature of MDA is the usage of models, i.e. the software development process is driven by modelling software systems and generating code or code fragments from these models.

8.2.1.1 The MDA Development Life Cycle

The MDA development life cycle is shown in figure 8.1. A significant difference to traditional development processes lies in the different artifacts that are created during

[1] The following section is mainly taken from [KWB04].

the different phases. According to MDA, most of the artifacts are formal models, of which the following three are most important.

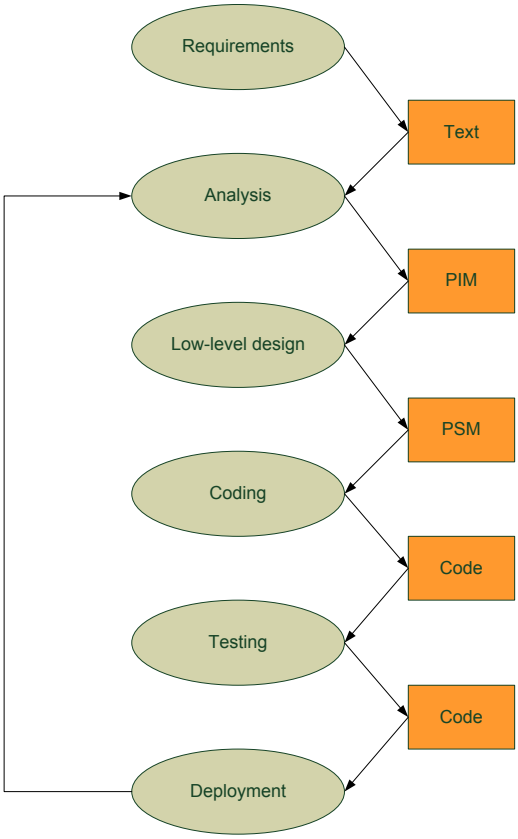


Figure 8.1: MDA development process

Platform Independent Model (PIM) The PIM is a model on a high level of abstraction, independent of any technology - be it a mainframe system with relational databases or an EJB application server. The system is modeled from the viewpoint of how it best supports the business that has to be supported by the software system that is developed, disregarding any technical details.

Platform Specific Model (PSM) The PIM is then transformed into a platform specific model (PSM), specifying the system in terms that are available in a particular implementation technology. A PSM for an JEE-based system would for instance contain EJB specific terms like “session bean” and “entity bean”, while a PSM for a

relational database system might include terms like “table”, “column” or “foreign key”.

For each target technology, an individual PSM has to be created. However, in practice, the PSM is usually omitted and PIMs are directly transformed into the next type of model, i.e. into code.

Code Actual code or at least code fragments are the final artifacts during the MDA development process, representing the executable form of the PIM that has initially been designed.

The key concept behind the three types of models is the increase of abstraction from lower levels to higher levels: developers are thus able to work on a higher level of abstraction, thus being able to cope with more complex systems with less effort.

8.2.1.2 Automation of the Transformation Steps

Further on, the crucial difference between MDA and traditional software development is the transformation between different types of models: while in traditional processes the transformations are mainly done by hand, MDA automates these steps by automatically generating a PSM from a PIM, and Code from the PSM. Current tools, such as AndroMDA (which will be presented in the following), are able to generate executable code from a high-level model specification at the click of a button. This leads to several major benefits of MDA:

- Since PIM developers do not have to cope with technical details, but focus on the business modelling instead, they can pay more attention to solving the business problems. More important, a large part of technical code (e.g. accessing databases, checking authorizations, ...) is automatically generated and does not have to be written by hand, leading to much higher *productivity*.
- Due to the fact that PIMs are (by definition) platform-independent, one single PIM can be transformed into PSMs (or code) for a variety of target platforms. Everything specified on the PIM level is completely portable. Since transformation tools are available for a variety of target platforms (or can additionally be self-developed, if necessary), this results in higher *portability* of the developed software system than with traditional development processes.

- A PIM implicitly fulfils the function of a high-level documentation for the software system. In addition, the PIM is not abandoned after writing: instead, any changed requirements are worked into the PIM (instead of simply changing code in traditional approaches), so that the parts of the application that have to be changed can be re-generated from the modified PIM. In addition, several tools allow the extraction of PIMs or PSMs from (legacy) applications, so that the quality of *maintenance and documentation* can significantly be improved by using MDA.

These benefits of MDA are not only claimed by the OMG; in fact, experience shows that the usage of MDA can significantly improve the software development process. For an in-depth evaluation of MDA in practice, we refer to the work of Pastor and Molina [PM07]. The applicability of MDA in several fields of applications has also extensively been evaluated, for instance for the integration of learning management systems [GBD05], the generation of database access applications [RLS05] and web applications [PH03], schema integration [KGF06, QKC05] or data warehousing [DL05, LMTS02, MTSP05].

8.2.1.3 Building Blocks of MDA

The following building blocks constitute the heart of MDA: the different models and the language they are written in, the transformation rules between different kinds of models together with a language in which to write those transformations, and the tools that execute the model transformations.

Models and Modelling Languages Any kind of model *describes a particular system* and *is written in a particular language*, and so are models in the MDA context. Although, by definition, MDA is not restricted to a particular model formalism, MDA models are usually written using UML [Obje].

In the following, we will only describe those aspects of UML that are important for our application of MDA; we assume that the reader has basic knowledge about UML. For detailed information, cf. the official UML specification [Obj04b, Objd, Obje] or appropriate textbooks [KWB04, Fra03, MSUW04] and papers, like [KdM05], to name but a few.

UML contains several different types of diagrams, e.g. class diagrams, sequence diagrams, statechart diagrams, etc. For our approach, only class diagrams are relevant.

Class diagrams mainly consist of classes (containing attributes) and associations between those classes. Figure 8.2 shows a sample class diagram.

Figure 8.2: Sample class diagram in UML

Although UML offers a big variety of diagram types and many constructs for these diagrams, it is usually necessary to extend UML in order to introduce new constructs that are necessary for the business case modelling. In other words, a specific meta-model has to be developed based on the default UML possibilities.

A meta-model is defined as a model of a model, specifying the language for all possible models, so that each model can be seen as an instance of the meta-model. This is visualized in figure 8.3: a model is written in a particular language, which is defined by its corresponding meta-model.

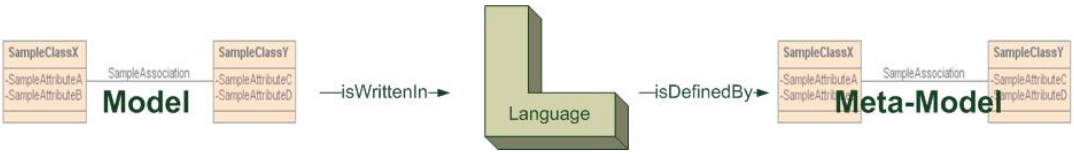


Figure 8.3: Model, language and meta-model

One way to extend UML and build a meta-model is called *Meta Object Facility (MOF)*. We are not going to highlight MOF in this dissertation; for an overview cf. the official website [Obja], containing the detailed MOF specification.

A second way for the construction of meta-models for UML are *UML profiles*. Due to the broad support for UML profiles in numerous modelling tools, such as *Magic Draw* [NoM], *Poseidon UML* [Gen], *Microsoft Visio Professional* [Mica], and many more, we decided to use UML profiles, which we will describe in the following.

UML profiles are defined as an "extension mechanism that can be used to customize UML for different platforms and domains without supporting a complete metamodelling capability" [Obj04a]. In contrast to the *heavyweight* MOF, UML profiles constitute a *lightweight* mechanism which is integrated into UML and its meta-model. Technically, UML profiles are collections of adaptions of the UML constructs, tailored

to specific business case relevant needs. According to the OMG, a UML profile is "a stereotyped package that contains model elements that have been customized for a specific domain or purpose using extension mechanisms, such as stereotypes, tagged definitions and constraints" [Obj04a]. Summarized very briefly, a UML profile can be used to define specializations of the basic UML constructs (like classes, attributes, etc.), using *stereotypes* and *tagged values*. Thus, a language for arbitrary UML models, suited to the specific business needs, can be defined. By referencing to such a UML profile, any UML model can make use of the new language constructs.

We will illustrate this by a short example. Let us assume that developers should be able to design models containing special classes that represent persistable entities. In addition, it should be possible to specify an attribute *tableName* for every persistable entity, naming the database table the entity should be stored in. A profile defining this language is visualized in figure 8.4: persistable entities can be marked by the stereotype *DatabaseEntity*. They are a specialization of standard UML classes (denoted by the keyword "Class" in brackets) with an additional attribute *tableName*.

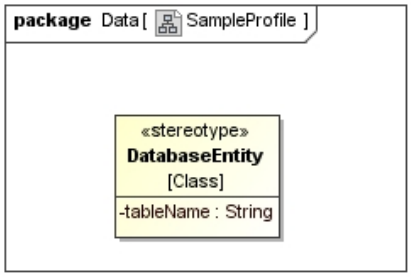


Figure 8.4: Sample UML profile

This profile can then be used in any UML model, i.e. the model may contain classes that are stereotyped as *DatabaseEntity*. Each of those classes can be further detailed by specifying the value of the attribute *tableName*. This value is called *tagged value* and denoted by the attribute name together with the attribute value. A sample UML class diagram using the profile from figure 8.4 is shown in figure 8.5.

Similar to classes, all existing UML constructs can be specialized using stereotypes. In our implementation, we will extend UML classes and attributes (associations between classes are a special form of attributes since UML 2.0) to define a meta-model for our event-handling system, as we will show in chapter 10.

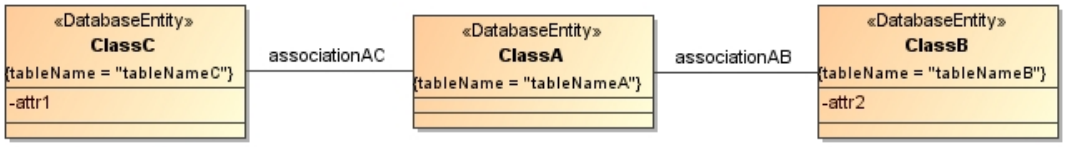


Figure 8.5: Sample UML model using a UML profile

Transformations and Transformation Languages An arbitrary model, built using a predefined meta-model, can then be given to a transformation as an input. Transformations are processes automatically converting one model to another model of the same system; in our case, they transform a platform independent model (PIM) into executable code. Therefore, transformation rules, written in a particular transformation language, are necessary: they define how one or more constructs in the source language (i.e. the model) can be transformed into one or more constructs in the target language (i.e. the code). Transformation rules are usually defined using *templates*: transformation tools take templates as an input and fill them with actual data from the source model, deriving the target source code. A sample template, generating a DDL-file to create a database instance for a UML model using the sample profile from figure 8.4 is shown in listing 8.1.

```
#foreach{$entity in $databaseEntities}
-- creating table for $entity
CREATE TABLE $entity.tableName;
#end
```

Listing 8.1: Sample transformation template

To achieve a mapping between stereotypes and the respective templates, a transformation tool then has to be configured so that it passes the set of all classes tagged as ≪DatabaseEntity≫ to the transformation. As a result, the transformation would yield the DDL file shown in listing 8.2.

```
-- creating table for ClassA
CREATE TABLE tableNameA;
-- creating table for ClassB
CREATE TABLE tableNameB;
-- creating table for ClassC
CREATE TABLE tableNameC;
```

Listing 8.2: Sample transformation result

Depending on the template language and the tool that is used to transform models, many different programming constructs can be used within the templates: for-each statements, conditional sections, even calls to complex functions that may be written in a high-level programming language can be issued from a template. In chapter 10, we will present the templates that generate the database triggers for our event-handling system realized using *AndroMDA*, a common MDA-tool that is briefly introduced in the next section.

8.2.2 AndroMDA

AndroMDA [Andb] is an open-source model-to-code transformator. UML models, which can be enhanced using one or more UML profiles, have to be designed with an arbitrary modelling tool, such as Magic Draw [NoM], and saved using the exchange format *XMI* (XML Metadata Interchange) [Objf]. The transformations from models to code are implemented using *metafacades*, *templates* and *configuration files*:

- *Metafacades* represent the stereotyped model elements during the transformation. For each stereotype in a UML profile, a corresponding metafacade-class has to exist. At runtime, AndroMDA creates one instance of the metafacade class for every UML element that is tagged with this stereotype. These instances can then be used within the transformation templates, for instance by accessing the metafacade's attributes (i.e. the tagged values in the UML model) or by calling method logic that has been implemented by the metafacade developers.
- *Templates* are the blueprints for the code that has to be generated. AndroMDA uses the template scripting language Velocity [Apa] to define templates. Besides the extensive language constructs offered by Velocity, metafacade methods can be called from Velocity templates, so that any algorithm that is needed to generate code can be implemented in the Java programming language.
- *Configuration files* finally bring together UML profiles, metafacades and templates: developers can configure the mapping between stereotypes and the respective metafacade classes as well as the mapping between metafacades and templates.

By running AndroMDA, an arbitrary UML model that has been extended using a UML profile can thus be transformed into code, as long as the respective metafacades, templates and configuration files have been developed.

To provide a solution to our problem, we developed these artifacts to generate database triggers and materialized views, as we will show in the following chapters. Before we will do so, the foundations of those technologies are briefly presented.

8.3 Active Database Technology

As an extension to relational database systems, active database technology provides the ability to react to updates of the stored data. This capability is traditionally used for a variety of applications: enforcing integrity constraints, monitoring and alerting, checking authorizations, maintaining views, and many more. The desired reaction to detected updates is specified using event-condition-action rules (ECA-rules):

- The specification of the *event* describes the type of update the system should respond to,
- the *condition* defines a constraint that must be fulfilled for the action to be executed
- and the *action* finally states the reaction to the detected modification.

In the following, we will only present the capabilities of active database technology that are important for our solution; for a detailed overview of active database systems, cf. [WC96] or [CCW00].

Most commercial database systems provide constructs to create such ECA rules, called *triggers*. Triggers are user-defined procedures that are automatically started by the database management system if the specified conditions are fulfilled. The (simplified) syntax to create triggers in DB2 [IBM] is shown in listing 8.3:

Since triggers are not part of the SQL-92 standard, but have been introduced with SQL:1999, the syntax may vary between different vendors. However, all database systems provide similar constructs:

- Each trigger is given an identifying name, allowing to modify or delete a created trigger by accessing it using its name.
- The trigger action time states whether the action has to be executed before or after the detected modification operation.

- The trigger event is tied to a table. Only modifications of data stored in this table are monitored by the trigger. A trigger can either react to insert, update or delete operations. Additionally, the monitoring can be limited to one or more columns of this table.
- Using the REFERENCING clause, the body of the trigger (i.e. the trigger action) can access the content of the modified table or the updated column as it was BEFORE the detected update or AFTER it.
- If a detected update modifies several rows, the trigger can either be fired once (FOR EACH STATEMENT) or once per updated row (FOR EACH ROW)
- The condition under which the trigger action should be executed can optionally be specified using standard SQL conditions.
- Finally, the action that should be performed as a reaction to the detected update can be specified in the triggered SQL statement. The full vocabulary of SQL is available; i.e. the action can be described using data modification language constructs, such as INSERT, UPDATE or DELETE statements, as well as by for instance calling stored procedures for more complex actions.

```
<DB2-trigger> ::= CREATE TRIGGER <trigger-name>
                  <trigger-action-time>
                  <trigger-event> ON <table-name>
                  [ REFERENCING <references> ]
                  <trigger-granularity>
                  [ <trigger-condition> ]
                  <triggered-SQL-statement>

<trigger-action-time> ::= BEFORE | AFTER

<trigger-event> ::= INSERT | DELETE | UPDATE [ OF <column-name> ]

<references> ::= OLD AS <identifier> | NEW AS <identifier>
                 OLD_TABLE AS <identifier> | NEW_TABLE AS <identifier>

<trigger-granularity> ::= FOR EACH { ROW | STATEMENT }

<trigger-condition> ::= WHEN ( <SQL-condition> )
```

Listing 8.3: Trigger syntax in DB2

As an example, let us assume that triggers should be responsible of storing the number of rows in tableA in the attribute count of table tableB. This can be achieved by providing two triggers, listening for insert- and delete-statements on tableA and updating tableB. Listing 8.4 shows these sample triggers.

As this example shows, triggers can - amongst other use cases - be used to keep computed derivations of the current data up to date. A more sophisticated technology that has been developed for this requirement are *materialized views*, which will be presented in the following.

```
-- listen to inserts and increase count
CREATE TRIGGER insertListener
AFTER INSERT ON tableA
UPDATE tableB
  SET tableB.count = tableB.count + 1;

-- listen to deletes and decrease count
CREATE TRIGGER deleteListener
AFTER DELETE ON tableA
UPDATE tableB
  SET tableB.count = tableB.count - 1;
```

Listing 8.4: Sample triggers

8.4 Materialized Views

Views in relational databases offer "virtual" relations, showing only an excerpt of the data model. In this context, virtual means that the definition of a view does not create new tables; instead, the content of the views is computed for every query accessing the view. Views are defined by specifying SQL queries which are evaluated every time the view is accessed. Listing 8.5 shows a sample view definition together with the data definition statement to create the underlying tables.

```
-- create base tables
CREATE TABLE tableA (
  attributeA Integer,
  attributeB Integer,
  attributeC Varchar );

CREATE TABLE tableB (
  attributeX Integer,
  attributeY Varchar,
  attributeZ Varchar );

-- create view for join between tableA and tableB
CREATE VIEW myView AS
  SELECT attributeB, attributeC, attributeZ
  FROM   tableA, tableB
  WHERE  tableA.attributeA = tableB.attributeX;
```

Listing 8.5: Sample view definition

Views can be used in queries instead of tables. Everytime the view is accessed, the result of the underlying query is computed. Listing 8.6 shows a possible usage of myView.

```
-- statement with direct table access
SELECT attributeB, attributeC, attributeZ
  FROM   tableA, tableB
  WHERE  tableA.attributeA = tableB.attributeX;

-- equivalent statement using predefined view
SELECT * from myView;
```

Listing 8.6: Sample view usage

While "traditional" views are re-computed during every query, many database systems offer the possibility to define *materialized views*. Views can be materialized by storing the tuples of the view in the database, so that, for instance, index structures can be built upon the materialized view. As a consequence, database accesses to these materialized views can be by far faster than accesses to views that are computed at runtime. Thus, a materialized view is like a cache - a (possibly aggregated) copy of the data [GM99b].

As a drawback, just like a cache, data stored in materialized views can become "dirty" when the tuples of the underlying base relations are updated, so that the view content has to be recomputed. The process of updating a materialized view in response to updates of the underlying data is called *view maintenance* and causes additional costs for write accesses to the base relations of a view. However, most current database systems support so-called *incremental view maintenance*, i.e. they do not re-compute the whole view content from scratch, but try to modify only those view fragments that are affected by an update, thus reducing computation effort and speeding up view maintenance [GM99b].

As a sample database system, DB2 [IBM] supports materialized views. To underline the fact that materialized view contents are actually stored in database tables, the syntax (shown in listing 8.7) is similar to the creation of tables.

```
<DB2-materialized-view> ::= CREATE TABLE <view-name> AS (<sql-query>)
                            [ <intial-defer-option> ]
                            [ <refresh-options> ]

<initial-defer-option> ::= DATA INITIALLY DEFERRED

<refresh-option> ::= REFRESH { DEFERRED | IMMEDIATE }
```

Listing 8.7: Simplified syntax to create materialized views in DB2

In addition to the definition of the materialized view by specifying the underlying SQL query, it is possible to define *when* the contents of the view have to be updated:

- If the option DATA INITIALLY DEFERRED is issued, the contents of the view are not computed at the time the view is created, but at the first time a query tries to read the contents of the materialized view.
- If the option REFRESH IMMEDIATE is specified, the contents of the view are updated every time the underlying base relations are updated.
- In contrast, if the contrary option REFRESH DEFERRED is specified, the contents of the view are recomputed only if a query accesses the view data.

Depending on the actual usage scenario of the view and the underlying base relations, this may lead to improvements (or deteriorations) of the runtime performance. In brief, if the base relations are updated often but the view is accessed seldomly, deferred view maintenance should lead to better results than immediate refreshment.

A last fact worth mentioning about materialized views are their limitations: depending on the database system, the definition of materialized views can be restricted. For instance, several database systems forbid view definitions that contain recursive or self-joining queries due to performance reasons. As we will show when presenting our solution, we had to find work-arounds for several of these shortcomes.

At this point, we only wanted to give a very brief overview of materialized views; for a detailed description, traditional fields of applications, techniques to maintain materialized views etc., cf. [GM99a].

8.5 Summary

This chapter tried to give an overview of the technologies we selected for our prototypic implementation of a model-driven event-handling framework, namely *Model Driven Architecture (MDA)*, *Active Database Systems* and *Materialized Views*. The basic concepts of these technologies which are needed to understand our solution were presented together with references to detailed information.

In the following chapter, we will present how these technologies are used and combined to realize a generic, generative and non-invasive event-handling framework.

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."

C.A.R. Hoare

9 Reference Architecture and Implementation

In chapter 4, we introduced the generic architecture of our approach and the respective components that have to be implemented. In the following, we will present an actual implementation of this architecture, using the previously introduced technologies: MDA for the specification of the event-handling semantics and the generation process, active database technology for the detection and processing of updates and materialized views for the storage of event propagation indices, i.e. for optimization purposes. A short summary concludes this chapter.

9.1 Substantiating the Abstraction Layers

In chapter 4, the different layers of abstraction (cf. figure 4.4) were presented. In the following, we will substantiate the *implementation abstraction* and present our UML profiles, the transformations of models into triggers as well as the optimization approach using materialized views. As we showed in chapter 5, our solution is based on a formal representation of the information system's data model and the event-handling semantics. Thus, for the approach to be applicable, a language is needed to

specify data and event-handling models that can be interpreted by the transformation tools to generate event-handling triggers. In our approach, we use UML profiles to provide such a modelling language, containing all concepts that are needed to define the database design, the event-handling constructs and the probability model. This language, an extension of the standard UML concepts (as shown in chapter 8), is then used by information system designers to create appropriate models of event-handling systems. In the following, we will describe the different aspects we designed into our profile, while the profile itself will be presented as a whole in chapter 10.

9.1.1 UML Profiles and MDA

As we already motivated, we decided to use UML as a starting point for the description of the data model. UML innately provides constructs to represent classes, attributes and associations, so no further work has to be done to develop a language (i.e. a meta-model) for these concepts. As an extension to this basic model, a description of the database data model that contains all information that is required to derive the appropriate triggers and SQL statements is needed. Instead of re-inventing the wheel, we use the UML profile shipped with the AndroMDA Hibernate Cartridge [Anda]. This cartridge was initially developed to design UML models that are meant to be persisted to a relational database. Thus, the profile contains, amongst many others, stereotypes for persistable classes. Entities can be tagged to specify the name of the corresponding database table and attributes can be tagged to represent the name of the corresponding column. Additionally, many-to-many associations can also be tagged to specify which table the many-to-many association has to be stored in. These simple constructs provide enough information for the MDA transformations, i.e. the data model the triggers are based on is known well enough.

For the representation of the event-handling constructs that have been introduced in chapter 5, new stereotypes and tagged values had to be introduced, containing representations of the concepts *Subscribers*, *Subscribables*, *event-propagating associations* and *implicit subscriptions*. Since UML models do not offer possibilities to separate multiple stereotyped classes and attributes into different groups, all constructs that can be part of an overlay were given an additional tagged value, containing the id(s) of the overlay(s) the respective construct belongs to. Like that, we are able to group subscribables, event-propagating associations and implicit subscriptions into overlays, which completes the representation of the semantic possibilities we presented in chapter 5.

The optimization approach we introduced in chapter 7 needs - amongst other information - data about the cardinalities of the different entities. As we already showed, the respective data can either be collected from the actual information system or be specified by developers at designtime. For the designtime specification, tagged values that allow model developers to enter the expected number of entities that are contained in the information system have been introduced.

Finally, as a last aspect of the event-handling model, the update probabilities of classes and associations are required, too. As we already proposed in chapter 7, this information can either be collected during runtime or be specified at design time by developers. For the latter case (which we use), additional (optional) tagged values that can be applied to entities (i.e. classes stereotyped as entity) were introduced, representing the update probability of the respective table the class or many-to-many association is stored in. A complete designtime probability model can be represented this way.

9.1.2 Transforming Models into Triggers

For the transformation of enhanced models we use the MDA transformation tool AndroMDA, which was introduced in the previous chapter. As a result, we generate triggers that monitor updates of the information system and create notification items for all relevant subscribers. The triggers implicitly contain our optimization solution by using materialized views to store event propagation indices, i.e. fragments of all event-propagating paths. Additionally, triggers that are able to monitor the usage behaviour of the data, i.e. to determine the probabilities of read- and write-accesses to the data, can be created.

9.1.2.1 Using Materialized Views for Optimization Purposes

As we already showed in chapter 7, a possibility to improve the runtime behaviour of the event-handling system is to use event propagation indices. In our architecture, the functionality to determine relevant subscribers is encoded into the respective triggers. The SELECT statements of those triggers that compute the targets of the event propagation and the relevant subscribers partly access materialized views containing fragments of the event-propagation indices in a precomputed form. In section 9.2, we will show in detail how this is done.

9.1.2.2 Cost Model for Relational Databases

The cost model remains as a last part of the abstraction layers that have to be concretized. Since our prototypic implementation is based on relational database access, a cost model that resembles the actual costs has to be used. For ideal results, detailed insight about the costs that arise for the read- and write operations is required. Most commercial database systems offer interfaces to access the database optimizer directly and evaluate queries with respect to their expected costs. By querying the optimizer directly, the actual costs could be determined.

However, since subsection 7.8.5 yielded that the heuristic cost model from section 7.7.1 returns results that are close enough to the actual costs, we decided to use the heuristic cost model for our prototypic implementation. Since the cost model implementation is hidden behind a clearly defined interface in our prototypic implementation (taking the query as an input and returning the expected costs), it is easy to exchange this implementation and query the database optimizer instead.

With this brief overview about the prototypic architecture in mind, we will next take a closer look at the different components of our implementation.

9.2 System Architecture Components in Detail

Although the overall architecture has already been presented, it remains to show how the individual parts of the event-handling component, i.e. the target(s) of the generation process, have to be implemented. Due to the MDA development cycle, this step is known as creating a reference implementation: all fragments of the system that later have to be generated using appropriate transformations once have to implemented "by hand" so that one can abstract from this concrete implementation and divide it into fixed parts (this is what later on becomes templates) and variable parts (depending on the model of the system under development).

9.2.1 Event-Handling Data Access Layer

A prerequisite for any event-handling system is the possibility to store notifications. In addition, our approach includes the use of explicit subscriptions, so - in contrast to the implicit subscriptions inherent to the data model - these subscriptions have to be stored, too.

Both subscriptions and notifications are stored generically in a relational database, either in the same instance as the information system (if allowed by security policies) or in a different instance. As we thus cannot assume that the information system's data and the event-handling system's data are in the same instance, foreign keys between event-handling data and "real" data are not possible. Therefore, we use a generic data model that allows references without using foreign keys, but instead identifies tuples using unique IDs (which every legacy data tuple has to contain) and a unique identifier for the table the tuple is stored in, e.g. the table name.

This generic referencing mechanism is used to store subscribers and subscribables for explicit subscriptions as well as subscribers and updated tuples for notifications. Figure 9.1 shows the corresponding data model.

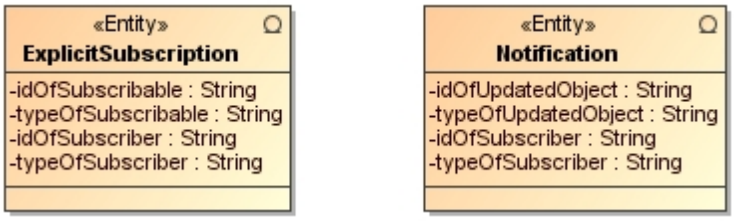


Figure 9.1: Data model for explicit subscriptions and notifications

If we recall the sample scenario from section 5.6, the explicit subscription between lectureA and maintainerM would thus be stored as a tuple

$$('lectureA',' Lectures',' maintainerM',' Maintainers')$$

in the database table that is used to store explicit subscriptions. Further, a notification for maintainerM after an update of lectureB would lead to the tuple

$$('lectureB',' Lectures',' maintainerM',' Maintainers')$$

in the database table for notifications.

By using this generic reference mechanism for both subscriptions and notifications, this part of the event-handling schema is generic and can be used for all use cases, i.e. for any arbitrary information system.

9.2.2 Event Processing using Triggers

In our architecture, triggers are used to monitor the database tables of the information system for modifications, automatically compute the subscribers that have to be informed about the detected update and store the respective notifications within the database, from where they can be presented to the subscribers, e.g. using a designated event-handling application with an appropriate GUI. This is visualized in figure 9.2.

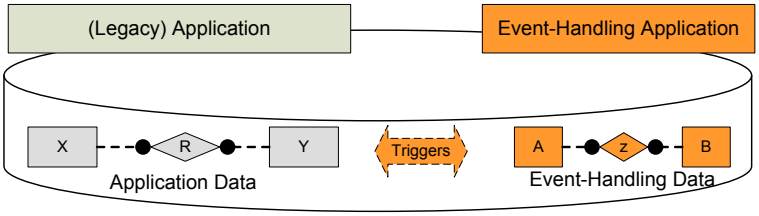


Figure 9.2: Architectural view: Triggers

This implies that the event-handling logic has to be coded into the database triggers. In the following, we will show how these triggers - which have to be generated from the event-handling model - are built. This implies a description of which tables to monitor, how to derive the respective subscriber and how to integrate our optimization approach.

9.2.2.1 Database Tables to Monitor

According to our concept (cf. chapter 5), every overlay in the event-handling model contains *one* subscribable that has to be monitored for modifications. However, in an implementation based on relational databases, it is not always sufficient to have only one trigger for the respective database table. Instead, we have to take a closer look at the monitored attribute(s) of the subscribable.[1]

Monitored Attribute is of Primitive Type Whenever a monitored attribute is of a primitive type, i.e. its value can be stored in a column of the database table, one UPDATE trigger for the respective column is enough. If a subscribable contains several monitored primitive attributes, they can share a common trigger containing the

[1] In the following, we only present one trigger for each scenario, detecting UPDATEs. Actually, three triggers that differ only in the monitored operation (UPDATE, INSERT, DELETE) are necessary whenever we talk about UPDATE triggers.

section "... AFTER UPDATE ON Subscribable OF <monitoredAttribute1>, ..., <monitoredAttributeN> ...".

The determination of the correct triggers is more complex if the monitored attribute is of a complex type, i.e. the value of the attribute is stored in a separate database table and associated to the subscribable using a foreign key reference. Figure 9.3 shows the different possibilities in case an attribute X is stored in a separate table.

Figure 9.3: Cardinalities for associated entities

In the following, these four situations are examined to find the attributes that have to be monitored to detect an update of an arbitrary attribute X's value.[2]

Monitored Attribute is of Complex Type with 1:1 Cardinality To examine this case, let Subscribable be connected to X via a 1:1 association. A relational design for this scenario is represented in figure 9.4.

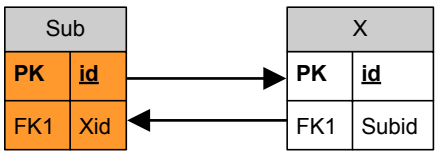


Figure 9.4: Relational model of 1:1 association

The value of Sub's attribute X can be caused by updating any attribute value of X, so any attribute of X has to be monitored by a trigger. In addition, the foreign key Subid must be observed, because a change of Subid indicates that X is assigned to a different Sub. As we can assume that the application handles foreign key references

[2]The graphical representations are simplified; for every scenario, we assume that there are attribute values in Sub and X that contain the actual data, in addition to the primary- and foreign keys that are depicted in the illustrations. Whenever we refer to "any attribute" of X or Sub, those additional attributes are meant.

correctly, we do not have to monitor Sub.Xid, because new assignments between Sub and X always lead to symmetric updates on both sides.

Monitored Attribute is of Complex Type with 1:n Cardinality This scenario is depicted in figure 9.5.

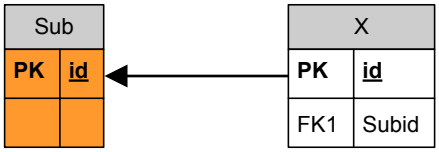


Figure 9.5: Relational model of 1:n association

Similar to the previous case, all attributes of X plus the foreign key references X.Subid have to be monitored to detect all modifications.

Monitored Attribute is of Complex Type with m:1 Cardinality Our third scenario is shown in figure 9.6.

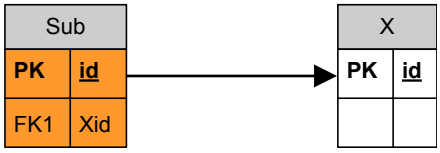


Figure 9.6: Relational model of m:1 association

Again, any attribute of X has to be observed for modifications. In addition to the previous cases, Sub has to be monitored too, since assignments of a new instance of X are handled by updating the foreign key reference Sub.Xid.

Monitored Attribute is of Complex Type with m:n Cardinality Finally, the most complex case, i.e. a $m : n$ association, is shown in figure 9.7.

In this situation, relationships between Sub and X are realized using a third table Join, holding foreign key references to Sub and X. Thus, Join has to be monitored for updates to detect new assignments. Further, any attribute of X, holding the actual content of X, must be monitored, too.[3]

[3] In practice, this situation becomes even more complex, since adding a new instance of X that is related to Subscribable leads to inserts on table X and table Join. Triggers fire immediately after an

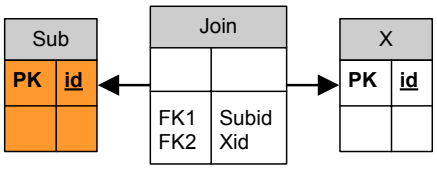


Figure 9.7: Relational model of m:n association

Table 9.2.2.1 finally collects these thoughts and shows which tables to monitor in a given scenario, and how to determine the id of the semantically updated instances of Sub if we assume that the respective trigger stores the reference to the updated tuple in a variable called new, while the old value of the updated tuple is represented as old.

Cardinality	Figure	Table(s) to monitor	Condition for updated Sub
1:1	9.4	X.all, X.Subid	Sub.id = new.Subid Sub.id = old.Subid
1:n	9.5	X.all, X.Subid	Sub.id = new.Subid Sub.id = old.Subid
m:1	9.6	X.all	Sub.Xid = new.id
		Sub.Xid	Sub.id = new.id
m:n	9.7	Join.all	Sub.id = new.Subid Sub.id = old.Subid
		X.all	Sub.id = Join.Subid AND Join.Xid = new.id Sub.id = Join.Subid AND Join.Xid = old.id

Table 9.1: Tables to monitor

To determine the overall set of tables that have to be monitored (i.e. the set of triggers that are necessary) regarding a subscribable, the following two rules must be followed:

- All observable attributes of Subscribable of a primitive type can be collected and commonly handled by one trigger. This trigger observes table Subscribable and all of its primitive-valued observed attributes.
- For each additional complex-typed observed attribute, up to four additional triggers (according to table 9.2.2.1) have to be used.[4]

update. However, to determine the respective instance of Subscribable the update "belongs to", the event-handling trigger has to see the new data after *both* updates. This can be achieved by asserting that both triggers are executed within the transaction context of the modifying transaction.

[4] As an optimization, it is possible to collect those triggers from table 9.2.2.1 that monitor an attribute of Subscribable and handle them analogously to primitive-typed attributes.

For illustration purposes, we take a look at the small data model depicted in figure 9.8. Let us assume that the entity Documents is the subscribable that has to be monitored by triggers. To be precise, the simple attribute content has to be observed, as well as the attribute belongsTo, i.e. the lectures that this document is used in. Let us further assume that the association between Documents and Lectures is a $m : n$ association.

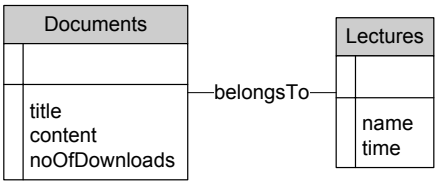


Figure 9.8: Subscribable entity with simple and complex observed attributes

In a normalized database design, the above-mentioned situation would be realized as shown in figure 9.9.

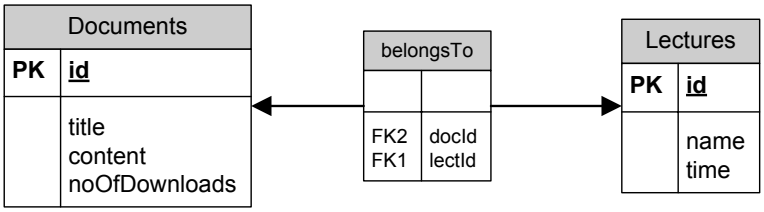


Figure 9.9: Normalized representation of entity with simple and complex attributes

In addition to the two entities, a third table belongsTo with foreign key references to Documents and Lectures is used to model the $m : n$ relationship. Thus, according to the results from table 9.2.2.1, a total of five triggers is required; one for the monitoring of the attribute content and four for the correct and complete monitoring of the complex attribute belongsTo. Listing 9.1 shows the resulting triggers in pseudo-SQL code.[5]

```
-- monitor simple attribute 'content'
CREATE TRIGGER trigger1
AFTER UPDATE OF content
ON documents
REFERENCING NEW AS new
--   handle update of document where
--   documents.id = new.id
;
```

[5]Again, we only present the AFTER UPDATE triggers for better readability. Additionally, AFTER INSERT and AFTER DELETE triggers are required, too.

```
-- monitor complex attribute 'belongsTo' (join table, new associated entity)
CREATE TRIGGER trigger2
AFTER UPDATE
ON belongsTo
REFERENCING NEW AS new
--   handle update of document where
--   documents.id = new.docId
;

-- monitor complex attribute 'belongsTo' (join table, old associated entity)
CREATE TRIGGER trigger3
AFTER UPDATE
ON belongsTo
REFERENCING OLD AS old
--   handle update of document where
--   documents.id = old.docId
;

-- monitor complex attribute 'belongsTo' (referenced table, new associated
    entity)
CREATE TRIGGER trigger4
AFTER UPDATE
ON lectures
REFERENCING NEW AS new
--   handle update of document where
--   documents.id = belongsTo.docId AND belongsTo.lectId = new.id
;

-- monitor complex attribute 'belongsTo' (referenced table, old associated
    entity)
CREATE TRIGGER trigger5
AFTER UPDATE
ON lectures
REFERENCING OLD AS old
--   handle update of document where
--   documents.id = belongsTo.docId AND belongsTo.lectId = old.id
;
```

Listing 9.1: Triggers to monitor attributes

In this listing, we only showed the structure of the triggers and how the actually update entity can be determined. In the following, we will present the "heart" of the triggers, i.e. the pseudo-SQL code of the reference implementation that determines all implicit and explicit subscribers and stores the respective notifications.

9.2.2.2 Determining Relevant Subscribers using Triggers

To actually determine all relevant subscribers (explicit and implicit) of an updated entity, the path descriptions for explicit and implicit subscribers, computed according

to algorithms 6.2 and 6.3 have to be known. Based on these path descriptions, the respective trigger content is constructed as follows:

Queries to Determine Explicit Subscribers For a sample path description from source Source, along subscribables Sub1 to SubN to target Target, the query to determine all explicit subscribers that have to be notified about an update of an instance of Source with id sourceId is shown in listing 9.2.

```
SELECT idOfSubscriber
FROM   ExplicitSubscription,
       Source,
       Sub1,
       ...,
       SubN,
       Target
WHERE  Source.id = sourceId
AND    Source.rightFK = Sub1.leftFK
AND    Sub1.rightFK   = Sub2.leftFK
AND    ... // follow the path description along all subscribables
AND    SubN.rightFK   = Target.leftFK
AND    Target.id      = ExplicitSubscription.idOfSubscribable
AND    ExplicitSubscription.typeOfSubscribable = 'Target'
;
```

Listing 9.2: Sample query to determine explicit subscribers

As we can see, this query simply evaluates the path description and finds all targets of paths respecting this description, and joins the result with all matching explicit subscriptions stored in table Subscription (cf. fig. 9.1).

As we will show in our example later on, queries like this one are then used in every monitoring trigger, i.e. for every explicit path description. Furthermore, for each path description, a set of n triggers (depending on the amount of tables to monitor, cf. section 9.2.2.1) is necessary.

Queries to Determine Implicit Subscribers The queries for implicit subscribers are built likewise. However, instead of joining the explicit subscription table, the join against the implicit subscriber is already contained in the path description (cf. alg. 6.3), so the id of the target of the path description represents the id of the subscriber.

Listing 9.3 shows such a query for a sample path description from source Source, via the subscribables Sub1 to SubN to target Target.

```
SELECT Target.id
FROM   Source,
       Sub1,
       ...,
       SubN,
       Target
WHERE  Source.id = sourceId
AND    Source.rightFK = Sub1.leftFK
AND    Sub1.rightFK   = Sub2.leftFK
AND    ... // follow the path description along all subscribables
AND    SubN.rightFK   = Target.leftFK
;
```

Listing 9.3: Sample query to determine implicit subscribers

Again, queries like this are used in every necessary trigger.

9.2.2.3 Example Without Optimization

To present a comprehensive example of the above-mentioned concepts, we recall the sample overlay from figure 5.11. Let us assume that the assocation between Lectures and Rooms is a $n : 1$ association, let us further assume that the reflexive association between Rooms is of cardinality $n : 1$ and that Rooms are maintained by several Maintainers and vice-versa, i.e. this association is of cardinality $m : n$. Finally, we assume that all simple attributes of Lectures are observed, as well as the complex attribute attends. This leads to the entity-relationship model as presented in figure 9.10.[6]

As we showed in section 6.3, the following implicit subscription path descriptions are computed (shown in simplified form)

Lectures, Rooms, Rooms, Rooms, Maintainers

Lectures, Rooms, Rooms, Maintainers

Lectures, Rooms, Maintainers

as well as the following explicit subscription paths:

Lectures, Rooms, Rooms, Rooms

[6]For better readability, we only show the relevant tables and omit entities Documents and belongsTo.

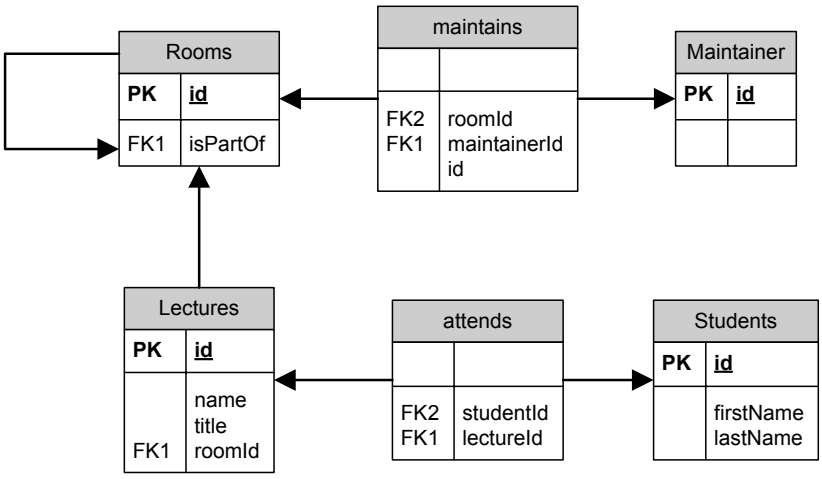


Figure 9.10: Sample entity-relationship model

Lectures, Rooms, Rooms

Lectures, Rooms

Lectures

This situation finally leads to the triggers shown in listing 9.4. To shorten the list, we only present all triggers monitoring Lectures' simple attributes completely. For the complex attribute observing triggers, we only present the trigger for the first implicit subscription path; the remaining triggers are built analogously. Furthermore, only AFTER UPDATE triggers are presented here, while AFTER INSERT and AFTER DELETE triggers are also necessary and have to be implemented analogously.

```
--
-- triggers monitoring Lectures' simple attributes
--

-- implicit path 1
CREATE TRIGGER  triggerImplicit1Simple
AFTER UPDATE OF name, time
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT         new.id,
                               'Lectures',
                               'Maintainers',
                               Maintainers.id
```

```
                FROM            Lectures,
                                Rooms r1,
                                Rooms r2,
                                Rooms r3,
                                maintains,
                                Maintainers
                WHERE           Lectures.id = new.id
                AND             Lectures.roomId = r1.id
                AND             r1.isPartOf = r2.id
                AND             r2.isPartOf = r3.id
                AND             r3.id = maintains.roomId
                AND             maintains.maintainerId = Maintainers.id
;

-- implicit path 2
CREATE TRIGGER  triggerImplicit2Simple
AFTER UPDATE OF name, time
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT          new.id,
                                'Lectures',
                                'Maintainers',
                                Maintainers.id
                FROM            Lectures,
                                Rooms r1,
                                Rooms r2,
                                maintains,
                                Maintainers
                WHERE           Lectures.id = new.id
                AND             Lectures.roomId = r1.id
                AND             r1.isPartOf = r2.id
                AND             r2.id = maintains.roomId
                AND             maintains.maintainerId = Maintainers.id
;

-- implicit path 3
CREATE TRIGGER  triggerImplicit3Simple
AFTER UPDATE OF name, time
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT          new.id,
                                'Lectures',
                                'Maintainers',
                                Maintainers.id
                FROM            Lectures,
                                Rooms r1,
                                maintains,
                                Maintainers
```

```
                WHERE           Lectures.id = new.id
                AND             Lectures.roomId = r1.id
                AND             r1.id = maintains.roomId
                AND             maintains.maintainerId = Maintainers.id
;

-- explicit path 1
CREATE TRIGGER  triggerExplicit1Simple
AFTER UPDATE OF name, time
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT         new.id,
                               'Lectures',
                               'ExplicitSubscriber',
                               ExplicitSubscription.idOfSubscriber
                FROM           Lectures,
                               Rooms r1,
                               Rooms r2,
                               Rooms r3,
                               ExplicitSubscription
                WHERE          Lectures.id = new.id
                AND            Lectures.roomId = r1.id
                AND            r1.isPartOf = r2.id
                AND            r2.isPartOf = r3.id
                AND            ExplicitSubscription.idOfSubscribable = r3.id
                AND            ExplicitSubscription.typeOfSubscribable = 'Rooms'
;

-- explicit path 2
CREATE TRIGGER  triggerExplicit2Simple
AFTER UPDATE OF name, time
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT         new.id,
                               'Lectures',
                               'ExplicitSubscriber',
                               ExplicitSubscription.idOfSubscriber
                FROM           Lectures,
                               Rooms r1,
                               Rooms r2,
                               ExplicitSubscription
                WHERE          Lectures.id = new.id
                AND            Lectures.roomId = r1.id
                AND            r1.isPartOf = r2.id
                AND            ExplicitSubscription.idOfSubscribable = r2.id
                AND            ExplicitSubscription.typeOfSubscribable = 'Rooms'
;
```

```
-- explicit path 3
CREATE TRIGGER  triggerExplicit3Simple
AFTER UPDATE OF name, time
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT         new.id,
                               'Lectures',
                               'ExplicitSubscriber',
                               ExplicitSubscription.idOfSubscriber
                FROM           Lectures,
                               Rooms r1, ExplicitSubscription
                WHERE          Lectures.id = new.id
                AND            Lectures.roomId = r1.id
                AND            ExplicitSubscription.idOfSubscribable = r1.id
                AND            ExplicitSubscription.typeOfSubscribable = 'Rooms'
;

-- explicit path 4
CREATE TRIGGER  triggerExplicit4Simple
AFTER UPDATE OF name, time
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT         new.id,
                               'Lectures',
                               'ExplicitSubscriber',
                               ExplicitSubscription.idOfSubscriber
                FROM           Lectures,
                               ExplicitSubscription
                WHERE          ExplicitSubscription.idOfSubscribable = new.id
                AND            ExplicitSubscription.typeOfSubscribable = 'Rooms'
;

--
-- triggers monitoring Lectures' complex attribute 'attends'
--
-- implicit path 1
CREATE TRIGGER  triggerImplicit1Complex1
AFTER UPDATE OF studentId, lectureId
ON              attends
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT         new.id,
                               'Lectures',
                               'Maintainers',
                               Maintainers.id
```

```
                FROM            Lectures,
                                Rooms r1,
                                Rooms r2,
                                Rooms r3,
                                maintains,
                                Maintainers,
                                attends
                WHERE           attends.studentId = new.studentId
                AND             Lectures.id = attends.lectureId
                AND             Lectures.roomId = r1.id
                AND             r1.isPartOf = r2.id
                AND             r2.isPartOf = r3.id
                AND             r3.id = maintains.roomId
                AND             maintains.maintainerId = Maintainers.id
;

CREATE TRIGGER  triggerImplicit1Complex2
AFTER UPDATE OF studentId, lectureId
ON              attends
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT          new.id,
                                'Lectures',
                                'Maintainers',
                                Maintainers.id
                FROM            Lectures,
                                Rooms r1,
                                Rooms r2,
                                Rooms r3,
                                maintains,
                                Maintainers,
                                attends
                WHERE           attends.studentId = new.studentId
                AND             Lectures.id = attends.lectureId
                AND             Lectures.roomId = r1.id
                AND             r1.isPartOf = r2.id
                AND             r2.isPartOf = r3.id
                AND             r3.id = maintains.roomId
                AND             maintains.maintainerId = Maintainers.id
;

...

-- implicit path 2
CREATE TRIGGER  triggerImplicit2Complex1
...
```

Listing 9.4: Triggers for sample entity-relationship model

By using a set of triggers like the ones shown in listing 9.4 for every overlay, the implicit and explicit subscribers can be determined and stored in table Notifications, from where they can be read and processed afterwards.

9.2.2.4 Processing Notification Entries

Due to the structure of the event-handling triggers, notifications are uniformously stored in table Notifications. Further on, it is very likely that this table contains duplicate entries (e.g. if a Maintainer is responsible for more than one Room and at least two of his or her rooms are affected by an update of Lectures). Although this is no ideal solution, we accept this minor deficit and solve the problem of redundant entries by using SELECT DISTINCT statements to determine all notifications for a particular user, as the following listing shows:

```
--
-- read notifications for subscriber with id <subId>
--
SELECT DISTINCT *
FROM   Notifications
WHERE  Notifications.idOfSubscriber = <subId>
;
```

Listing 9.5: SQL query to read notifications for a particular user

Although the resulting notifications are correct with respect to the concept presented in part II, as we will show in chapter 10, there is still optimization potential, which we will explain in the following.

9.2.2.5 Optimization

Two aspects of our approach can further be optimized with respect to the runtime behaviour: first of all, distinct triggers reacting to the same event (i.e. with the same trigger header) can be combined into one trigger; second, the optimization approach from chapter 7 has to be integrated into the trigger code.

Grouping Triggers with the Same Header Instead of using up to n triggers reacting to the same update event and separately evaluating n path descriptions, these triggers can be grouped into one trigger using the SQL UNION statement to collect all results.

Listing 9.6 shows (in abbreviated form) what our example from listing 9.4 looks like with consideration of this optimization.

```
--
-- triggers monitoring Lectures' simple attributes
--

-- implicit path 1
CREATE TRIGGER  triggerImplicit1Simple
AFTER UPDATE OF name, time
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT         new.id,
                               'Lectures',
                               'Maintainers',
                               Maintainers.id
                FROM           Lectures,
                               Rooms r1,
                               Rooms r2,
                               Rooms r3,
                               maintains,
                               Maintainers
                WHERE          Lectures.id = new.id
                AND            Lectures.roomId = r1.id
                AND            r1.isPartOf = r2.id
                AND            r2.isPartOf = r3.id
                AND            r3.id = maintains.roomId
                AND            maintains.maintainerId = Maintainers.id
                UNION
                SELECT         new.id,
                               'Lectures',
                               'Maintainers',
                               Maintainers.id
                FROM           Lectures,
                               Rooms r1,
                               Rooms r2,
                               maintains,
                               Maintainers
                WHERE          Lectures.id = new.id
                AND            Lectures.roomId = r1.id
                AND            r1.isPartOf = r2.id
                AND            r2.id = maintains.roomId
                AND            maintains.maintainerId = Maintainers.id
                UNION
                SELECT         new.id,
                               'Lectures',
                               'Maintainers',
                               Maintainers.id
```

```
                FROM            Lectures,
                                Rooms r1,
                                maintains,
                                Maintainers
                WHERE           Lectures.id = new.id
                AND             Lectures.roomId = r1.id
                AND             r1.id = maintains.roomId
                AND             maintains.maintainerId = Maintainers.id
;

-- explicit path 1
CREATE TRIGGER  triggerExplicit1Simple
AFTER UPDATE OF name, time
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT          new.id,
                                'Lectures',
                                'ExplicitSubscriber',
                                ExplicitSubscription.idOfSubscriber
                FROM            Lectures,
                                Rooms r1,
                                Rooms r2,
                                Rooms r3,
                                ExplicitSubscription
                WHERE           Lectures.id = new.id
                AND             Lectures.roomId = r1.id
                AND             r1.isPartOf = r2.id
                AND             r2.isPartOf = r3.id
                AND             ExplicitSubscription.idOfSubscribable = r3.id
                AND             ExplicitSubscription.typeOfSubscribable = 'Rooms'
                UNION
                SELECT          new.id,
                                'Lectures',
                                'ExplicitSubscriber',
                                ExplicitSubscription.idOfSubscriber
                FROM            Lectures,
                                Rooms r1,
                                Rooms r2,
                                ExplicitSubscription
                WHERE           Lectures.id = new.id
                AND             Lectures.roomId = r1.id
                AND             r1.isPartOf = r2.id
                AND             ExplicitSubscription.idOfSubscribable = r2.id
                AND             ExplicitSubscription.typeOfSubscribable = 'Rooms'
                UNION
                SELECT          new.id,
                                'Lectures',
                                'ExplicitSubscriber',
                                ExplicitSubscription.idOfSubscriber
                FROM            Lectures,
                                Rooms r1,
                                ExplicitSubscription
```

```
                WHERE           Lectures.id = new.id
                AND             Lectures.roomId = r1.id
                AND             ExplicitSubscription.idOfSubscribable = r1.id
                AND             ExplicitSubscription.typeOfSubscribable = 'Rooms'
                UNION
                SELECT          new.id,
                                'Lectures',
                                'ExplicitSubscriber',
                                ExplicitSubscription.idOfSubscriber
                FROM            Lectures,
                                ExplicitSubscription
                WHERE           ExplicitSubscription.idOfSubscribable = new.id
                AND             ExplicitSubscription.typeOfSubscribable = 'Rooms'
;

--
-- triggers monitoring Lectures' complex attribute 'attends'
--

-- implicit path 1
CREATE TRIGGER  triggerImplicit1Complex1
AFTER UPDATE OF studentId, lectureId
ON              attends
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT          new.id,
                                'Lectures',
                                'Maintainers',
                                Maintainers.id
                FROM            Lectures,
                                Rooms r1,
                                Rooms r2,
                                Rooms r3,
                                maintains,
                                Maintainers,
                                attends
                WHERE           attends.studentId = new.studentId
                AND             Lectures.id = attends.lectureId
                AND             Lectures.roomId = r1.id
                AND             r1.isPartOf = r2.id
                AND             r2.isPartOf = r3.id
                AND             r3.id = maintains.roomId
                AND             maintains.maintainerId = Maintainers.id
                UNION
                SELECT          new.id,
                                'Lectures',
                                'Maintainers',
                                Maintainers.id
```

```
                FROM            Lectures,
                                Rooms r1,
                                Rooms r2,
                                Rooms r3,
                                maintains,
                                Maintainers,
                                attends
                WHERE           attends.studentId = new.studentId
                AND             Lectures.id = attends.lectureId
                AND             Lectures.roomId = r1.id
                AND             r1.isPartOf = r2.id
                AND             r2.isPartOf = r3.id
                AND             r3.id = maintains.roomId
                AND             maintains.maintainerId = Maintainers.id
                UNION ...

...

-- implicit path 2
CREATE TRIGGER  triggerImplicit2Complex1
...
```

Listing 9.6: Triggers for sample entity-relationship model using SQL UNION

Materialized Views as Event-Propagation Indices In addition to the previously presented technical optimization, it remains to show how the optimization approach presented in chapter 7 has been integrated into our reference implementation.

Due to our optimization approach, for every relevant path description an optimal partitioning into regular path fragments and event propagation indices can be determined. Looking at the path

Lectures, Rooms, Rooms, Rooms, Maintainers

let us assume that the optimization algorithm revealed that an optimal solution is to use an event propagation index index for the fragment Rooms, Rooms, Rooms. In an implementation for active database technology, this can be realized by creating a materialized view that precomputes all paths matching this path description. Listing 9.7 shows a simplified SQL statement to create this view.[7]

[7] Actually, commercial database systems are unable to automatically maintain materialized views with self-joins. Therefore, in appendix B we show how index maintenance can be realized using separate triggers.

```
--
-- materialized view to precompute paths matching the
-- description Rooms, Rooms, Rooms
--
CREATE TABLE index
AS
SELECT r1.id AS leftId, r3.id AS rightId
FROM   Rooms r1, Rooms r2, Rooms r3
WHERE  r1.isPartOf = r2.id
  AND  r2.isPartOf = r3.id
DATA INITIALLY DEFERRED
REFRESH IMMEDIATE
;
```

Listing 9.7: Sample statement to create a materialized view as an event propagation index

Views are then integrated into the triggers' queries to determine implicit and explicit subscribers. Listing 9.8 shows the query for path

Lectures, Rooms, Rooms, Rooms, Maintainers

using the materialized view from listing 9.7 as an event propagation index.

```
--
-- query to determine all paths matching the path description
-- Lectures, Rooms, Rooms, Rooms, Maintainers
--
SELECT Maintainers.id
FROM Lectures, index, Maintainers, maintains
WHERE Lectures.id = <updated id>
  AND Lectures.roomId = index.leftId
  AND index.rightId = maintains.roomId
  AND maintains.maintainerId = Maintainers.id
;
```

Listing 9.8: Query to determine subscribers, using event propagation index

Assuming that two event propagation index views index3Rooms (for the path-description fragment Rooms, Rooms, Rooms) and index2Rooms (for the path-description fragment Rooms, Rooms) are available, the trigger code from listing 9.6 is shown in listing 9.9.

```
--
-- triggers monitoring Lectures' simple attributes
--

-- implicit path 1
CREATE TRIGGER  triggerImplicit1Simple
AFTER UPDATE OF name, time
```

```
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT         new.id,
                              'Lectures',
                              'Maintainers',
                              Maintainers.id
                FROM          Lectures,
                              index3Rooms,
                              maintains,
                              Maintainers
                WHERE         Lectures.id = new.id
                AND           Lectures.roomId = index3Rooms.leftId
                AND           index3Rooms.rightId = maintains.roomId
                AND           maintains.maintainerId = Maintainers.id
                UNION
                SELECT        new.id,
                              'Lectures',
                              'Maintainers',
                              Maintainers.id
                FROM          Lectures,
                              index2Rooms,
                              maintains,
                              Maintainers
                WHERE         Lectures.id = new.id
                AND           Lectures.roomId = index2Rooms.leftId
                AND           index2Rooms.rightId = maintains.roomId
                AND           maintains.maintainerId = Maintainers.id
                UNION
                SELECT        new.id,
                              'Lectures',
                              'Maintainers',
                              Maintainers.id
                FROM          Lectures,
                              Rooms r1,
                              maintains,
                              Maintainers
                WHERE         Lectures.id = new.id
                AND           Lectures.roomId = r1.id
                AND           r1.id = maintains.roomId
                AND           maintains.maintainerId = Maintainers.id
;

-- explicit path 1
CREATE TRIGGER  triggerExplicit1Simple
AFTER UPDATE OF name, time
ON              Lectures
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
                               idOfSubscriber)
                SELECT         new.id,
                              'Lectures',
```

```
                                'ExplicitSubscriber',
                                ExplicitSubscription.idOfSubscriber
                FROM            Lectures,
                                index3Rooms,
                                ExplicitSubscription
                WHERE           Lectures.id = new.id
                AND             Lectures.roomId = index3Rooms.leftId
                AND             index3Rooms.rightId = ExplicitSubscription.
                    idOfSubscribable
                AND             ExplicitSubscription.typeOfSubscribable = 'Rooms'
                UNION
                SELECT          new.id,
                                'Lectures',
                                'ExplicitSubscriber',
                                ExplicitSubscription.idOfSubscriber
                FROM            Lectures,
                                index2Rooms,
                                ExplicitSubscription
                WHERE           Lectures.id = new.id
                AND             Lectures.roomId = index2Rooms.leftId
                AND             index2Rooms.rightId = ExplicitSubscription.
                    idOfSubscribable
                AND             ExplicitSubscription.typeOfSubscribable = 'Rooms'
                UNION
                SELECT          new.id,
                                'Lectures',
                                'ExplicitSubscriber',
                                ExplicitSubscription.idOfSubscriber
                FROM            Lectures,
                                Rooms r1, ExplicitSubscription
                WHERE           Lectures.id = new.id
                AND             Lectures.roomId = r1.id
                AND             ExplicitSubscription.idOfSubscribable = r1.id
                AND             ExplicitSubscription.typeOfSubscribable = 'Rooms'
                UNION
                SELECT          new.id,
                                'Lectures',
                                'ExplicitSubscriber',
                                ExplicitSubscription.idOfSubscriber
                FROM            Lectures,
                                ExplicitSubscription
                WHERE           ExplicitSubscription.idOfSubscribable = new.id
                AND             ExplicitSubscription.typeOfSubscribable = 'Rooms'
;

--
-- triggers monitoring Lectures' complex attribute 'attends'
--
-- implicit path 1
CREATE TRIGGER  triggerImplicit1Complex1
AFTER UPDATE OF studentId, lectureId
ON              attends
INSERT INTO     notifications (idOfUpdatedObject,
                               typeOfUpdatedObject,
                               typeOfSubscriber,
```

```
                                idOfSubscriber)
                SELECT          new.id,
                                'Lectures',
                                'Maintainers',
                                Maintainers.id
                FROM            Lectures,
                                index3Rooms,
                                maintains,
                                Maintainers,
                                attends
                WHERE           attends.studentId = new.studentId
                AND             Lectures.id = attends.lectureId
                AND             Lectures.roomId = index3Rooms.leftId
                AND             index3Rooms.rightId = maintains.roomId
                AND             maintains.maintainerId = Maintainers.id
                UNION
                SELECT          new.id,
                                'Lectures',
                                'Maintainers',
                                Maintainers.id
                FROM            Lectures,
                                index3Rooms,
                                maintains,
                                Maintainers,
                                attends
                WHERE           attends.studentId = new.studentId
                AND             Lectures.id = attends.lectureId
                AND             Lectures.roomId = index3Rooms.leftId
                AND             index3Rooms.rightId = maintains.roomId
                AND             maintains.maintainerId = Maintainers.id
                UNION ...

...
-- implicit path 2
CREATE TRIGGER  triggerImplicit2Complex1
...
```

Listing 9.9: Optimized triggers for sample entity-relationship model

This listing shows the final optimized version of triggers as they are used in our reference implementation.

9.2.2.6 Inheritance Revisited

In section 5.7.1, we briefly outlined how inheritance, a central concept of object-oriented modelling, can be integrated into our approach. In the following, we will briefly describe how inheritance could be handled in our prototypic implementation.

Object-oriented inheritance is no immanent concept of relational databases and thus has to be realized by using appropriate relational models which resemble the object-oriented inheritance. To overcome this deficit, also known as the *object-relational impedance mismatch*, different strategies are known (for a detailed discussion on the impedance mismatch, cf. [EN04]). If inheritance regarding a superclass A and several subclasses B1 to Bn has to be handled, the following four solutions are alternatively possible:

- For an arbitrary superclass A, a designated database table is designed, containing all attributes of A. Further, for all subclasses B1 to Bn, an additional table, containing the primary key of A (which is, due to the semantics of the inheritance, also primary key of B1 to Bn) and all additional attributes of the respective subclass, is necessary.
- For each subclass B, a designated table, containing all attributes of superclass A plus the attributes of the subclass B is created, but no designated table to store instances of superclass A is required.
- Only one database table is created, containing the union of all attributes of the superclass A and of all subclasses, plus a designated column to store the the type of the actual instance that is represented in this row.
- Similarly, onle one database containing the union of all attributes of all classes is created. In contrast to the third option, boolean flags indicate the membership of a tuple to any of the super- or subclasses, making this approach more suitable if overlapping subclassing has to be modeled.

One thing that is common to all of the four approaches is the fact that

- depending on the used approach, all tables that contain instances of any super- or subclass can clearly be determined,
- each of the necessary tables contains the primary key of the super- or subclass,
- and it can clearly be determined in which table the different attributes are contained.

Thus, triggers could automatically be generated, knowing which columns of which tables to monitor and where to find the attributes that are needed for the subscriber determination. Thus, it remains an open issue of our prototypic implementation to implement the support of inheritance, but the principal capability of our approach to deal with inheritance is clearly present.

9.3 Summary

This chapter presented the reference implementation for our event-handling approach, based on active database technology. We showed how the different abstraction layers were instantiated for this particular technology, how explicit subscriptions and notifications are modeled, and - most important - what the triggers that detect all updates and determine the relevant subscribers look like. As it is the purpose of a MDA reference implementation, all the artifacts that are identified within this exemplary realization are used as a blueprint for the automatic generation process. Thus, we will show in the next chapter how UML profiles and transformation templates are used to be able to automatically generate event-handling triggers from the respective information system model.

"The shortest route to getting things done is just do it."
Takayuki Ikkaku, Arisa Hosaka and Toshihiro Kawabata

10 From UML Models to Optimized Triggers

According to the MDA development paradigm, the step following the reference implementation is to design UML profiles that represent the meta-model. Next, transformation templates and transformation rules that are able to generate the actual runtime system have to be developed.

In this chapter, we will present the UML profile and the transformations into event-handling triggers. Furthermore, we will explain how the optimization approach has been realized. The chapter concludes with the proof that the generated triggers are correct, i.e. that they fulfil the requirements we postulated in chapter 6, followed by a comprehensive example.

10.1 UML Profile

The UML profile, i.e. the meta-model for the design of an arbitrary event-handling system, constitutes the heart of our model driven solution. In the following, we will present our UML profile for event-handling systems in detail.

As we have learned from our studies of the application domain, the knowledge that is required to specify all relevant aspects of the event-handling system can be divided

into three categories. First, information about the object-relational mapping, i.e. about how the object-oriented model is represented in its relational implementation, is needed. Second and most important, the event-handling constructs contained in our generic concept need a representation within the UML profile. These aspects implicitly constitute the domain specific language of our event-handling approach. Finally, as a possibility to pass statistic information to the optimizer, knowledge about the cardinalities of entities that are stored in the information system, as well as information about the expected update behaviour, has to be contained in the event-handling system's model and thus needs a representation in the meta-model.

Before we will illustrate the different aspects, we present a complete picture of the UML profile in figure 10.1.

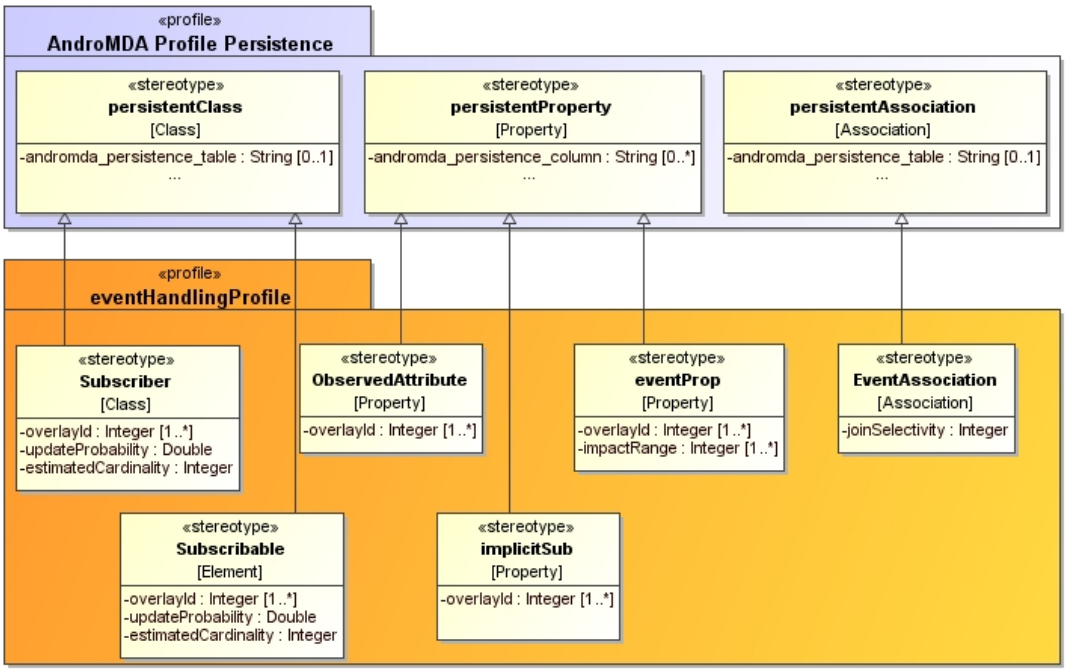


Figure 10.1: UML profile for the MDA implementation using active databases

10.1.1 Object-Relational Mapping

The object-relational mapping, i.e. the conversion from object-oriented design into relational database mapping is an important part of our approach: to generate correct

trigger code, it is necessary to know about the realization of the corresponding entities in the entity-relationship model. Table names, column names, foreign key constraints and the respective attributes and/or join tables have to be known for the trigger generation to work.

In our approach, we re-use the persistence profile supplied by AndroMDA [Andb]. In figure 10.1, these stereotypes are depicted in the blue profile in abbreviated form. The profile contains (amongst others) the following stereotypes containing a multitude of attributes, of which we will show only those that are important for our solution:

- Stereotype ≪persistentClass≫
  Instances of classes tagged with this stereotype are designed to have persistent representations in the relational database.

  - Attribute andromda_persistence_table
    The value of this attribute specifies which database table the instances have to be stored in.

- Stereotype ≪persistentProperty≫
  A persistent property denotes a persistent attribute of a persistent class, i.e. an attribute that has to be stored in a column of the persistent classes table. In contrast, attributes without this tag are designed to be transient, i.e. they are for instance derived from persistent attributes at runtime.

  According to UML 2.0, associations between classes are also treated as properties, i.e. this stereotype can also be applied to association ends, thus representing the column in which a foreign key reference is stored.

  - Attribute andromda_persistence_column
    This attribute specifies the name of the column the attribute (or foreign key reference) should be stored in.

- Stereotype ≪persistentAssociation≫
  Associations that should be persisted can be tagged with this stereotype. It is solely used for $m : n$ associations, i.e. associations that need a join table in their relational representation.

  - Attribute andromda_persistence_table
    Similar to persistent classes, this attribute specifies the name of the table the foreign keys for the $m : n$ association are stored in.

Although the above-mentioned stereotypes denote only a small excerpt from the possibilities AndroMDA offers, they constitute the basis on which the event-handling profile elements are built on. We will present these stereotypes in the following.

10.1.2 Event-Handling Profile Elements

All of the event-handling profile elements are specializations of the persistence elements we presented in section 10.1.1. They combine a representation of the event-handling constructs we introduced in chapter 5 and of the runtime behaviour specification as proposed in chapter 7. In detail, the following stereotypes are available for use in our profile eventHandlingProfile:

- Stereotype ≪Subscriber≫
  According to our concept, subscribers represent the potential receivers of update notifications. The stereotype ≪Subscriber≫ can be used to tag all classes in the information system model which actually should be interpreted as a subscriber.

  - Attribute overlayId
    In our concept, overlays play an important part. To specify which overlay(s) a particular subscriber belongs to, the attribute overlayId is used. Due to its cardinality (one or more values) we can assert that every subscriber belongs to at least one overlay.

  - Attribute updateProbability
    For our optimization approach, the update probability of all subscribable classes is required. If this information should already be specified at design-time (as proposed in section 7.6.2), it can be specified via the float value updateProbability.

  - Attribute estimatedCardinality
    Finally, the cost model needs information about the cardinality, i.e. the number of instances of this class that are persisted to the database. To represent this information at design time, the attribute estimatedCardinality can be used.

- Stereotype ≪Subscribable≫
  Similar to subscribers, this stereotype can be used to denote all subscribables in an event-handling model. This stereotype's attributes are equal to the attributes of subscribables, so they are not explained again. In addition, the attribute

overlayId denotes the id of the overlay in which this subscribable is the source subscribable.

- Stereotype ≪ObservedAttribute≫
  Since we do not want to monitor all attributes of a subscribable but only selected ones, it is necessary to offer a possibility to tag those attributes that have to be monitored. This can be done using the stereotype ≪ObservedAttribute≫, which can be applied to any persistent property and thus also to any association end (to observe complex attributes as introduced in the previous chapter).

  - Attribute overlayId
    Subscribables can be part of several overlays. In addition, in different contexts (i.e. in different overlays) it is possible that different attributes have to be observed. Therefore, one or more overlayIds can be specified for any observed attribute.

- Stereotype ≪eventProp≫
  For another major construct in our concept, event-propagating associations, this stereotype can be used. It has to be applied to association ends, which, in the UML jargon, are persistent properties, too.[1] By tagging association ends instead of associations, it is possible to specify the direction of the event propagation: the untagged end of an association represents the source, while the tagged end represents the target of the event propagation.

  - Attribute overlayId
    As for all of our stereotypes, this attribute represents the overlay assignment(s).

  - Attribute impactRange
    The attribute impactRange is used to represent the propagation distance of an update event, as proposed by our concept. Since event propagating associations may have different impact ranges in different overlays, the impact range is specified as an array, containing the impact ranges for all assigned overlays.

- Stereotype ≪implicitSub≫
  For specifying implicit subscription, the stereotype ≪implicitSub≫ is provided.

[1] To limit this stereotype's usage to association ends only, according OCL constraints would be necessary. However, we do not regard any constraints in this chapter, but define an open end of our work in chapter 12 instead.

It is used similarly to ≪eventProp≫: the tagged association end represents the direction of the implicit subscription.

  - Attribute overlayId
    Again, this attribute denotes the overlay assignment(s).

- Stereotype ≪EventAssociation≫
  Finally, for the cost model to work properly, it is necessary to have information about the join selectivity of associations. To be able to specify this, the respective associations can be tagged as an ≪EventAssocation≫.

  - Attribute joinSelectivity
    This attribute contains information about the actual join selectivity of an association, so that this value can be specified at designtime, if required.

Using these stereotypes, all of our event-handling constructs have a representation in our UML profile and can thus be applied to any (new or legacy) information system model.

10.2 Model Template

Besides the dynamic parts of an event-handling information system, a static part containing the data model for explicit subscriptions and notifications is required. As we already explained in the previous chapter, these entities are modeled using the above-mentioned stereotypes for persistent entities. By providing a UML template on which the actual information system can be built, information system designers are supported during this task. Figure 10.2 shows this template.

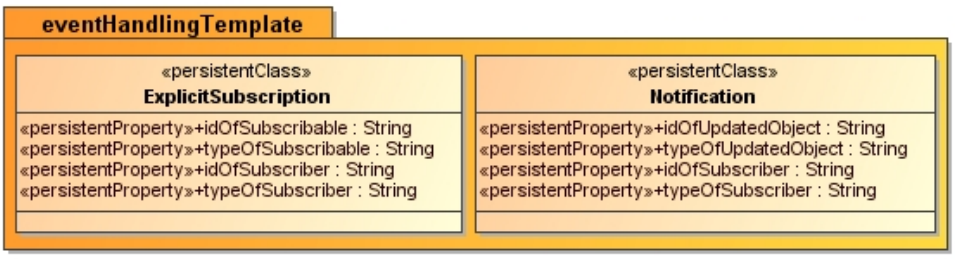


Figure 10.2: UML template model containing explicit subscriptions and notifications

Like this, we make sure that the tables for explicit subscriptions and notifications are present in every information system's data model, and thus the generated triggers work correctly.

10.3 Metafacades and Transformation Helper Classes

Profiles contain the stereotypes and tagged values that are required to enhance static models at design time. To transform this static information into artifacts like, in our case, database triggers, additional transformation logic is needed. As for AndroMDA, this logic is coded in so-called metafacades; classes that represent model elements and contain additional logic. The methods of the metafacades, in which this logic is contained, can then be called (from the transformation templates) during the model-to-code transformation.

According to good object-oriented design, these metafacades realize their transformation logic using additional helper classes. These classes are no metafacades themselves, but are created and used from within the metafacades' coding.

The metafacades that have been developed for our solution and the helper classes they make use of are depicted in figure 10.3. The diagram shows a simplified view of the implemented classes, just to give an idea of how the transformation has been built.[2]

10.3.1 AndroMDA Base Metafacades

The metafacades we describe in the following are part of the AndroMDA persistence cartridge. As they correspond the persistence profile (which we reuse, too) and contain methods to access all persistence-related information, they are used as a basis for our event-handling metafacades, which are derived from these base classes.

- Metafacade Entity
  This metafacade class is instantiated once per class in the UML model which is stereotyped as Entity and allows access to its properties.

[2]For instance, all getter-methods that allow access to the actual values of the stereotypes' attributes, for instance a method getImpactRange() to determine the tagged value impactRange of an association end that has been stereotyped as event propagating) are not shown in this figure.

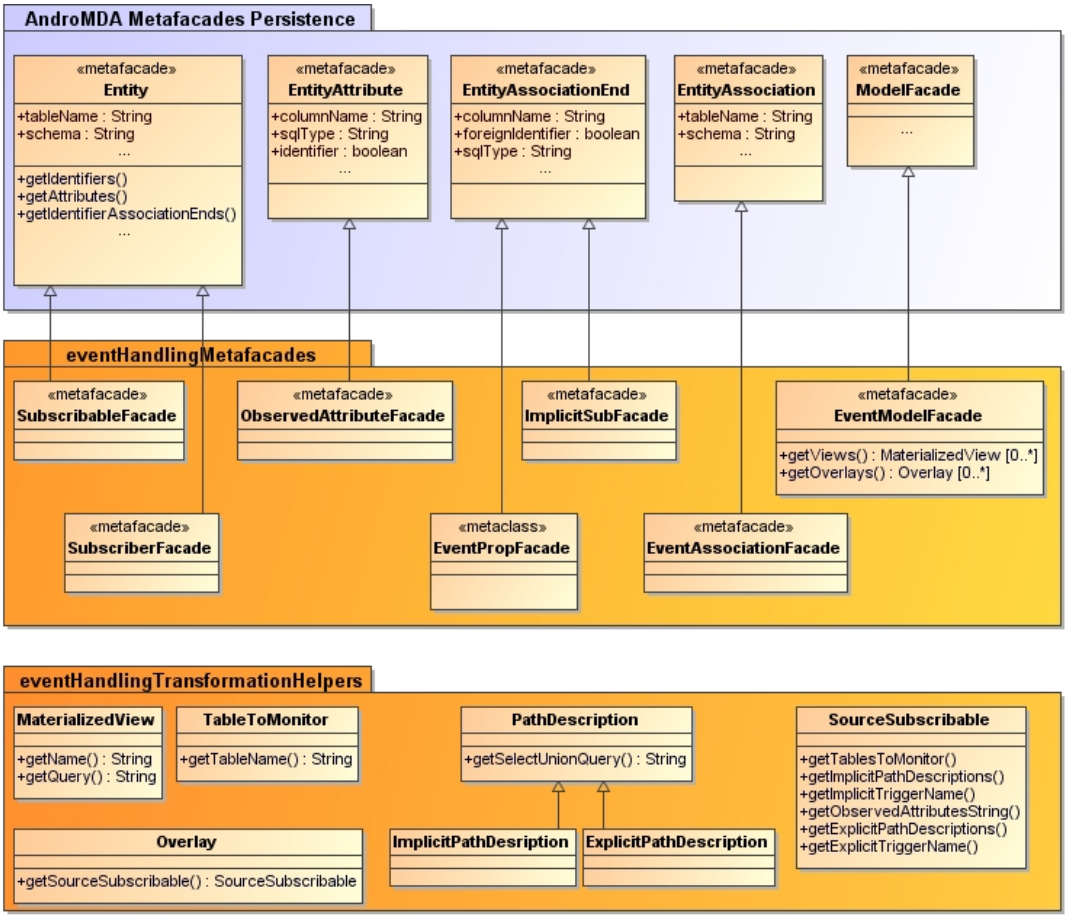


Figure 10.3: Metafacades and helper classes for model-to-code transformation

- Attribute tableName
  This attribute holds the name of the table in which actual entities in the UML model are stored.

- Attribute schema
  The schema name of the respective database instance is kept in this attribute.

- Method getIdentifiers()
  Using this method, a collection containing all identifying attributes of this

entity, i.e. the primary keys in the respective relational model, can be obtained. The method returns a collection of EntityAttributes.

  - Method getAttributes()
    Analogously, this method returns all attributes of the corresponding entity as a collection of EntityAttributes, including also the primary keys.

  - Method getIdentifierAssociationEnds()
    Since primary keys in the underlying entity-relationship model do not necessarily have to be simple attributes, but can also be associated entities (modeled in UML by associations), this method can be used to return all foreign keys to associated entities which act as identifiers.

- Metafacade EntityAttribute
  As already mentioned, this class facades access to all model elements that are stereotyped as an entity's attributes.

  - Attribute columnName
    This attribute holds the name of the column in which the attribute's values are stored in the relational model.

  - Attribute sqlType
    This attribute represents the SQL type of the attribute, for instance INTEGER or VARCHAR.

  - Attribute identifier
    Finally, this attribute holds a boolean flag whether the attribute is an identifier or not.

- Metafacade EntityAssociationEnd
  According to the UML standard, both ends of associations between classes can be accessed separately from the association itself. Thus, the metafacade EntityAssociationEnd represents one end of any association between two entities.

  - Attribute columnName
    This attribute contains the name of the column in which the association end is stored, i.e. the name of the respective foreign key column.

  - Attribute foreignIdentifier
    If the foreign key column is part of an entity's primary key, this can be found out by accessing this flag.

  - Attribute sqlType
    Similar to simple EntityAttributes, the SQL type of the foreign key column is stored in this attribute.

- Metafacade EntityAssociation
  In AndroMDA, associations between entities are stored using instances of the metafacade class EntityAssociation. Since all information that is needed to handle $1:1$, $1:n$ and $m:1$ associations is available from the respective association ends, this metafacade is required only for $m:n$ associations, as the metafacade's attributes reveal:

  - Attribute tableName
    This attribute holds the name of the table in which the $m:n$ association is stored, i.e. of the join table holding the foreign key references to both associated entities.

  - Attribute schema
    Additionally, the name of the schema in which the association is stored is contained in this attribute.

- ModelFacade
  Finally, the metafacade ModelFacade represents a UML model as a whole.

10.3.2 Event-Handling Metafacades

The transformations we implemented use the following subclasses of the AndroMDA metafacades, inheriting all database-relevant information from their parent classes. As mentioned above, getter-methods to access the tagged values are omitted in this metafacade presentation. Instead, we name the respective stereotype in the UML profile and thus imply that all tagged values of this stereotype are accessible via the metafacade.

- Metafacade SubscribableFacade
  This metafacade, inheriting from metafacade Entity and corresponding to stereotype «Subscribable», can be used to handle all of the UML model's classes that are tagged as subscribable and to access the respective tagged values overlayId, updateProbability and estimatedCardinality.

- Metafacade SubscriberFacade
  Like SubscribableFacade, this metafacade, also inheriting from metafacade Entity

and corresponding to stereotype ≪Subscriber≫, can be used to handle all classes that are tagged as subscriber and to access tagged values overlayId, updateProbability and estimatedCardinality.

- Metafacade ObservedAttributeFacade
  This metafacade, inheriting from EntityAttribute and corresponding to ≪ObservedAttribute≫ can be used to access observed attributes of subscribable entities, for instance to determine their overlayId.

- Metafacade EventPropFacade
  To access all event-propagating associations in the UML model, i.e. the association ends that are stereotyped with ≪eventProp≫, this metafacade has been designed. Inheriting from metafacade EntityAssociationEnd, all tagged values that describe the relational model of this association end are accessible; additionally, access to overlayId and impactRange is possible.

- Metafacade ImplicitSubFacade
  The metafacade ImplicitSubFacade has been designed similarly to EventPropFacade, except for the fact that only the tagged value overlayId is accessible.

- Metafacade EventAssociationFacade
  Counterpart to stereotype ≪EventAssociation≫, this metafacade represents all associations in the UML model that have been tagged with this stereotype. In addition to an EntityAssociationFacade's properties, access to the association's joinSelectivity is possible.

- Metafacade EventModelFacade
  Finally, the metafacade Model has been extended to represent any arbitrary UML model which has been enhanced with event-handling stereotypes. This metafacade is used by the transformations as an entry point to compute all required materialized views and all overlays contained in the model. This is the only metafacade with no corresponding stereotype in our UML profile.

  - Method getViews()
    This method computes all required materialized views (i.e. the optimization indices) according to the algorithms from chapter 7. The result is returned as a collection of MaterializedView instances; this helper class will be described in the next section.

  - Method getOverlays()
    As a second helper method, getOverlays() returns all overlays that are con-

tained in the UML model by evaluating all overlayIds from the different stereotyped classes. As a result, a collection of Overlay instances is returned.

10.3.3 Transformation Helper Classes

In line with good object-oriented design, the metafacade classes should not implement all transformation logic themselves, but rely on additional classes that represent parts of the transformation logic and contain appropriate business logic. Therefore, the following classes have been implemented and are used within the metafacades' method implementations:

- Class MaterializedView
  This helper class is used to represent materialized views which work as event propagation indices. Internally, an ordered list consisting of persistent entities and associations, representing path fragments in the UML model, is kept as an instance of PathDescription. This description is used to determine the definition of the materialized view.
  - Method getName()
    This method computes the name of the materialized view which is unique within the whole UML model.
  - Method getQuery()
    Using the internal list of connected persistent entities, together with the information about their relational representation (i.e. table names, primary keys, foreign keys, ...), a query that can be used in the definition of this materialized view is computed by method getQuery(), which internally uses PathDescription.getSelectUnionQuery() for this purpose.
- Class TableToMonitor
  Helper class TableToMontor is used to represent any table in the database model that needs to be monitored when observing a particular subscribable.
  - Method getTableName()
    This method simply returns the name of the database table that is represented by this helper class.
- Class Overlay
  According to our concept, all event-handling constructs are grouped into over-

lays. Helper class Overlay represents such an overlay, i.e. it groups all event-handling constructs having the same overlayId in the UML model.

  - Method getSourceSubscribable()
    Since overlays contain a designated subscribable that is monitored for changes, this method has been developed to return this source subscribable for an arbitrary overlay.

- Class PathDescription
  As already mentioned during the description of MaterializedView, descriptions of paths (and path fragments) are represented using helper class PathDescription. This class encapsulates an ordered collection of entities and their associations.

  - Method getSelectUnionQuery()
    This method returns a query in the form of SELECT ... UNION ... UNION, which can be used to compute all paths that match the represented path description.

- Class ImplicitPathDescription
  This subclass of PathDescription is used to describe implicit path descriptions.

- Class ExplicitPathDescription
  In contrast to ImplicitPathDescription, this subclass of PathDescription is used to describe explicit path descriptions.

- Class SourceSubscribable
  Finally, this class is used to represent a subscribable being the source of an overlay. Since overlays (and thus their source subscribable) represent the starting point for every event-handling trigger, several methods containing the event-handling transformation logic are realized within this class:

  - Method getTablesToMonitor()
    As we explained in section 9.2.2.1, a source subscribable may have several database tables that need to be monitored in order to record an update properly. Thus, this method computes and returns all instances of TableToMonitor that have to be monitored.

  - Method getImplicitPathDescriptions()
    All implicit path descriptions that are required to determine the implicit subscribers of an update are computed and returned by this method. It realizes algorithm 6.3 and returns the resulting path descriptions. Further-

more, the path description can contain event propagation indices instead of entity path elements, i.e. method getImplicitPathDescriptions() implicitly makes use of the optimization results that were obtained by method getViews() in class EventModelFacade.

- Method getImplicitTriggerName()
  As we will see in the next section, implicit path descriptions are evaluated within the triggers. Thus, a unique name for the implicit trigger is needed, which is computed by this method.

- Method getObservedAttributesString()
  Since only selected attributes of the source subscribable are to be monitored for changes, the respective trigger has to be limited to those attributes. Therefore, method getObservedAttributesAsString() returns the names of these attributes in a SQL conform manner.

- Method getExplicitPathDescriptions()
  Similarly to implicit path descriptions, the explicit path descriptions (again containing optimiziation information) are required. They are computed according to algorithm 6.2 by this method.

- Method getExplicitTriggerName()
  As a last method of this helper class, getExplicitTriggerName() returns a unique name for the trigger which is going to monitor the respective source subscribable and determine all of its explicit subscribers.

All of these metafacade and helper classes have been realized in Java, making use of the AndroMDA transformation framework. As we already mentioned, their purpose is to be called from the transformation template, which we will present in the following.

10.4 MDA Transformations

Within the AndroMDA framework, transformation templates are written using the Velocity template engine [Apa]. In conjunction with an appropriate configuration, these templates are evaluated once per UML model.

To generate the event-handling triggers, two templates form the building blocks of our solution; one for the materialized views working as event-propagating indices and one for the triggers themselves have been developed.

Listing 10.1 shows the template which is used to generate SQL code that creates the materialized views. It simply consists of a #foreach-loop iterating over all necessary materialized views as computed by method getViews() in the metafacade EventModelFacade. Inside the loop (ll. 7-10), the materialized view is created, using the helper class MaterializedView to compute the unique name and, most important, the SQL statement to define the underlying view.

```
-- create all required materialized views (event-propagating indices) which
-- are computed by the metafacade for the event-handling model
#foreach ($view in $model.getViews())
 -- view name and respective query are
 -- specific to every materialized view
 CREATE TABLE ${view.getName()}
 AS ${view.getQuery()}
 DATA INITIALLY DEFERRED
 REFRESH IMMEDIATE;

#end
```

Listing 10.1: Transformation template to create materialized views

As one can see (and as we tried to explain above), the important parts of the generation algorithm are coded in the metafacade- and helper classes. The same holds for template 10.2 which is used to generate all trigger SQL code. The outer #foreach-loop iterates over all overlays that are present in the UML model (computed by the respective helper-class). For each of the overlays, the corresponding source subscribable is determined (l. 3) - this iteration corresponds to the foreach-loop in algorithm 6.1. Based on this source subscribable, all tables that need to be monitored (cf. complex attributes as described in section 9.2.2.1) are considered and the inner part (ll. 10-49) is evaluated for each of the tables to monitor.

In this inner section, first all implicit path descriptions starting from the source subscribable are handled (ll. 10-28): for each implicit path description, one trigger determining all implicit subscribers and storing the respective notification in case of an UPDATE, as well as one analogous trigger for the detection of INSERTS are created. Each of those triggers only considers updates of observed attributes, as they are returned by the metafacade class SourceSubscribable, depending on the currently evaluated table to monitor.

In the second major part of the template (ll. 32-49), the same kind of trigger generation is coded, but for explicit subscribers, respectively.

```
-- iterate over all overlays that are part of the model
#foreach ($overlay in $model.getOverlays())
 #set ($sourceSub = $overlay.getSourceSubscribable())

 -- iterate over all tables that need to be monitored
 #foreach ($ttm in ${sourceSub.getTablesToMonitor())

  -- determine all implicit path descriptions that need to be handled for
  -- this subscribable and iterate over them
  #foreach ($ip in ${sourceSub.getImplicitPathDescriptions())

   -- create trigger detecting updates, for the current implicit path
   -- description and the current table to monitor
   CREATE TRIGGER ${sourceSub.getImplicitTriggerName($ip)}U
   AFTER UPDATE OF ${sourceSub.getObservedAttributesString($ttm)}
   ON ${ttm.getTableName()}
   INSERT INTO notifications (idOfUpdatedObject, typeOfUpdatedObject,
                              typeOfSubscriber, idOfSubscriber)
   ${ip.getSelectUnionQuery($ttm)};

   -- create similar trigger detecting inserts
   CREATE TRIGGER ${sourceSub.getImplicitTriggerName($ip)}I
   AFTER INSERT OF ${sourceSub.getObservedAttributesString($ttm)}
   ON ${ttm.getTableName()}
   INSERT INTO notifications (idOfUpdatedObject, typeOfUpdatedObject,
                              typeOfSubscriber, idOfSubscriber)
   ${ip.getSelectUnionQuery($ttm)};
  #end

  -- determine all explicit path descriptions that need to be handled for
  -- this subscribable and iterate over them
  #foreach ($ep in ${sourceSub.getExplicitPathDescriptions())

   -- create trigger detecting updates, for the current explicit path
   -- description and the current table to monitor
   CREATE TRIGGER ${sourceSub.getExplicitTriggerName($ep)}U
   AFTER UPDATE OF ${sourceSub.getObservedAttributesString($ttm)}
   ON ${ttm.getTableName()}
   INSERT INTO notifications (idOfUpdatedObject, typeOfUpdatedObject,
                              typeOfSubscriber, idOfSubscriber)
   ${ep.getSelectUnionQuery($ttm)};

   -- create similar trigger detecting inserts
   CREATE TRIGGER ${sourceSub.getExplicitTriggerName($ep)}I
   AFTER INSERT OF ${sourceSub.getObservedAttributesString($ttm)}
   ON ${ttm.getTableName()}
   INSERT INTO notifications (idOfUpdatedObject, typeOfUpdatedObject,
                              typeOfSubscriber, idOfSubscriber)
   ${ep.getSelectUnionQuery($ttm)};
  #end
 #end
#end
```

Listing 10.2: Transformation template to create triggers

Of course, this description only reveals a brief overview of the realized metafacades, helper classes and templates. However, since the foundations of the underlying algorithms and techniques have already been presented in the previous chapters of this dissertation, we decided not to go into more detail here but instead focus on an easily understandable high-level software design. However, we briefly want to illustrate that this implementation actually generates triggers that are correct with respect to the definitions from chapter 5.

10.5 Correctness

In the following, we will present the key points of our implementation's correctness proof. As we showed in chapter 6, the presented trigger generation procedure is correct if

- **the generation algorithms from chapter 6 are implemented**
  Although we did not present the implementations of all metafacades' and helper classes' methods, we claim that the respective methods and templates implement the given algorithms correctly. For instance, the loop over all overlays in template 10.2 corresponds to the loop in algorithm 6.1. Furthermore, the algorithms to compute explicit and implicit paths have been implemented exactly as specified in chapter 6.

- **the explicit triggers are correct according to definition 6.2.2**
  The correctness of explicit triggers can easily be shown:

  1. Each of the triggers that are created using template 10.2 obviously monitors modifications of the source subscribable of the input paths - either by directly monitoring the respective table's observed attributes, or by monitoring one of the additional tables that have to be monitored. By creating two complementary triggers, one for all UPDATEs and one for the INSERTs, all relevant modifications are detected. Thus, condition 1 of definition 6.2.2 is fulfilled.

  2. As one can see from the example above, due to the created SELECT-statement within the view describing the path descriptions, the explicit subscribers of the path description targets are determined. Since all possible path descriptions are handled by method getSelectUnionQuery(), all targets of any path description are joined against explicit subscribers and thus condition 2 of definition 6.2.2 is fulfilled, too.

- **the implicit triggers are correct according to definition 6.2.3**
  The fulfilment of this requirement can be shown similarly to the previous requirement concerning explicit triggers and is thus left to the reader.

- **the combination of generated triggers forms a correct trigger combination with respect to definition 6.2.4**
  This final postulation can easily be proved: due to the additive character of the trigger action, a new entry is added to the notifications table for every detected update and every implicit or explicit subscriber. In combination with a duplicate-eliminating access to this notification table (SELECT DISTINCT ...), the union-set semantics which are required by definition 6.2.4 are provided.

Considering all these properties of the model-driven trigger generation together with the universal proof from chapter 6, we can postulate that our MDA solution creates triggers which are correct with respect to the underlying formal concept.

10.6 Comprehensive Sample Model

To complete this chapter describing our MDA solution, we end with a comprehensive sample model illustrating all of the above-mentioned concepts, profiles and stereotypes and showing the resulting SQL trigger code. The following example has been taken from a slightly adapted model of Stud.IP[3] and contains different event-handling overlays, fulfilling several event-handling requirements each.

The example is shown using different levels of abstraction: starting with the (slightly simplified) object-oriented UML model of the application, giving insight into the relational representation showing the appropriate entity-relationship model and finally presenting the database triggers that are created from the UML model using our transformation.

The following entities are part of the model:

- Studiengaenge represents all courses of study that are available
- AuthUserMd5 holds information about Stud.IP's users (i.e. students)
- Seminare contains the stored lectures

[3]Since Stud.IP uses the relational database system mySql which does not support foreign key constraints, the respective foreign key references have been added to the model manually.

- SeminarUser stores the assignment of students to lectures, enriched by an attribute holding the status of this assignment
- Dokumente represents the documents that have been stored and possibly assigned to their corresponding lectures
- Comments contains all comments that have been made concerning a particular document
- SemTree models a simple taxonomy of terms, so that lectures (Seminare) can be arranged hierarchically by assigning them to the respective element of the SemTree
- Institute finally represent all institutes (or chairs) within Stud.IP

Figure 10.4 shows the corresponding UML model. Furthermore, by using our event-handling profile, the following overlay information has been added to the model (presented in figure 10.5 which contains only the event-handling stereotypes):[4]

1. Every user shall be informed if one of the lectures (Seminare) he attends is modified.
2. Every user shall be informed if one of the documents assigned to one of the lectures he attends is modified.
3. Every user shall be informed if a category or a super-category of one of the lectures he attends is modified. Notifications shall only be created if the taxonomic relationship is at most two levels higher in the taxonomy.

According to AndroMDAs object relational mapping, this object-oriented model is transformed into the entity-relationship model depicted in figure 10.6.

Finally, applying the transformations we described in this chapter, we receive the listings 10.3 to 10.5 as a result.

```
---
-- view definitions for event-propagating indices
---

-----------
-- overlay1
-----------

```

[4]Due to a bug in the UML tool, the update probability values are displayed like string values, although they are actually of type double.

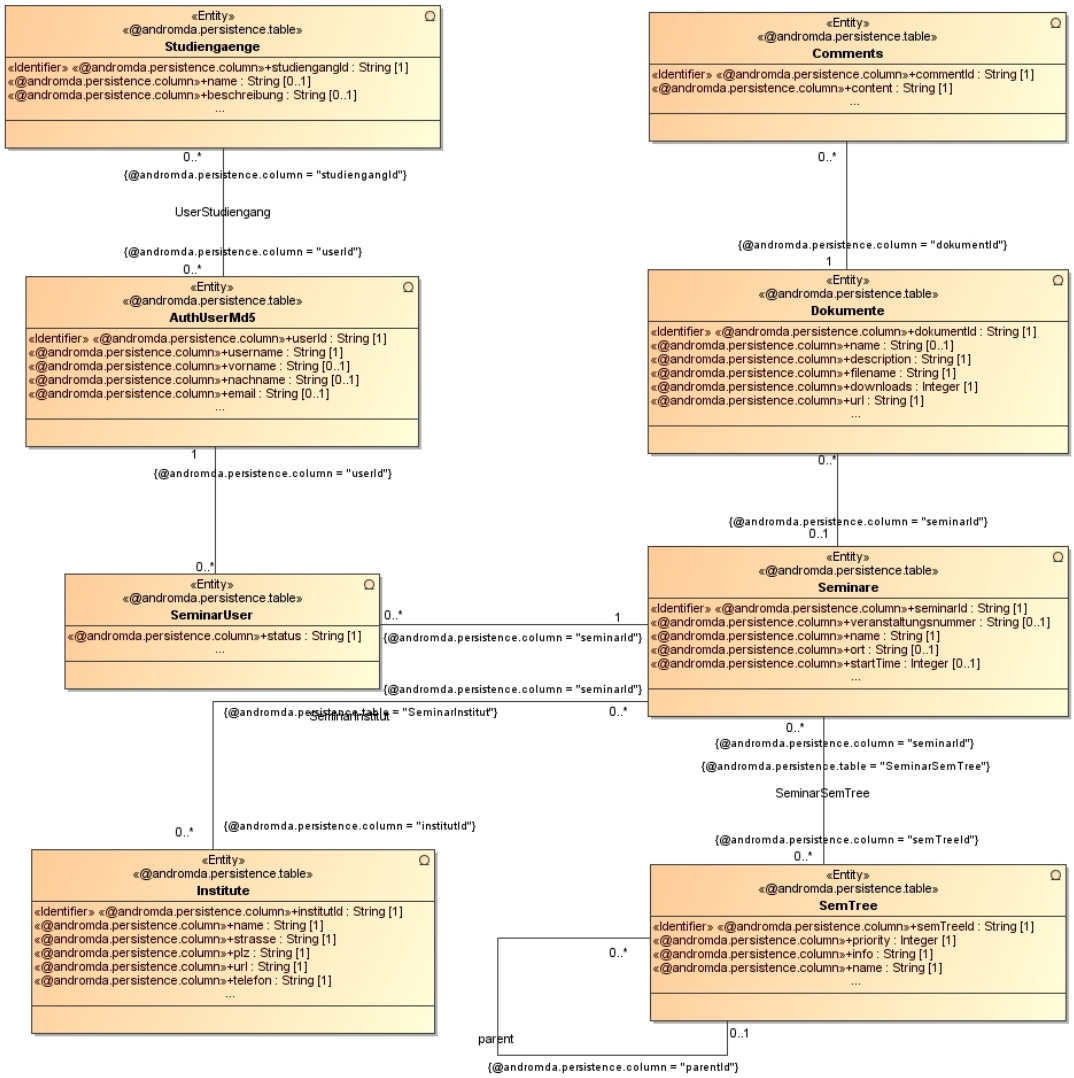


Figure 10.4: UML model of Stud.IP

```
-- view 1
CREATE TABLE view1
AS (SELECT SEMINARE.SEMINAR_ID AS left,
           AUTH_USER_MD5.USER_ID AS right
    FROM   SEMINARE,
           SEMINAR_USER,
           AUTH_USER_MD5
    WHERE  SEMINARE.SEMINAR_ID = SEMINAR_USER.SEMINAR_ID
    AND    SEMINAR_USER.USER_ID = AUTH_USER_MD5.USER_ID )
```

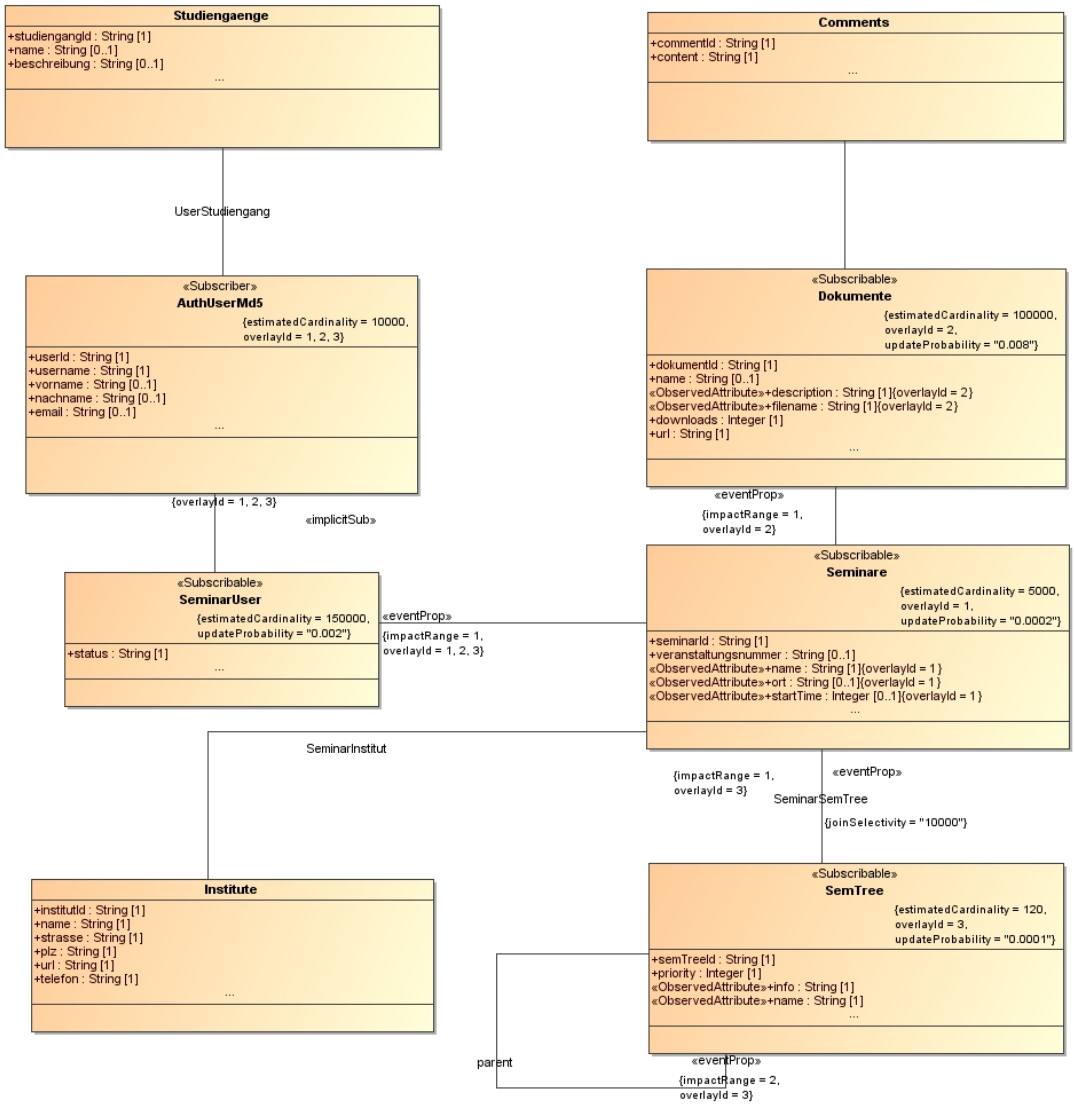


Figure 10.5: Stereotyped UML model for Stud.IP's event-handling

```
DATA INITIALLY DEFERRED
REFRESH DEFERRED;

-----------
-- overlay2
-----------

-- view2
```

```
CREATE TABLE view2
AS (SELECT SEMINARE.SEMINAR_ID AS left,
           AUTH_USER_MD5.USER_ID AS right
    FROM   SEMINARE,
           SEMINAR_USER,
           AUTH_USER_MD5
    WHERE  SEMINARE.SEMINAR_ID = SEMINAR_USER.SEMINAR_ID
    AND    SEMINAR_USER.USER_ID = AUTH_USER_MD5.USER_ID )
DATA INITIALLY DEFERRED
REFRESH DEFERRED;

-----------
-- overlay3
-----------

-- view3
CREATE TABLE view3
AS (SELECT SEM_TREE.SEM_TREE_ID AS left,
           AUTH_USER_MD5.USER_ID AS right
    FROM   SEM_TREE,
           SEMINAR_SEM_TREE,
           SEMINARE,
           SEMINAR_USER,
           AUTH_USER_MD5
    WHERE  SEM_TREE.SEM_TREE_ID = SEMINAR_SEM_TREE.SEM_TREE_ID
    AND    SEMINAR_SEM_TREE.SEMINAR_ID = SEMINARE.SEMINAR_ID
    AND    SEMINARE.SEMINAR_ID = SEMINAR_USER.SEMINAR_ID
    AND    SEMINAR_USER.USER_ID = AUTH_USER_MD5.USER_ID

    UNION

    SELECT st1.SEM_TREE_ID AS left,
           AUTH_USER_MD5.USER_ID AS right
    FROM   SEM_TREE st1,
           SEM_TREE st2,
           SEMINAR_SEM_TREE,
           SEMINARE,
           SEMINAR_USER,
           AUTH_USER_MD5
    WHERE  st1.SEM_TREE_ID = st2.PARENT_ID
    AND    st2.SEM_TREE_ID = SEMINAR_SEM_TREE.SEM_TREE_ID
    AND    SEMINAR_SEM_TREE.SEMINAR_ID = SEMINARE.SEMINAR_ID
    AND    SEMINARE.SEMINAR_ID = SEMINAR_USER.SEMINAR_ID
    AND    SEMINAR_USER.USER_ID = AUTH_USER_MD5.USER_ID

    UNION

    SELECT st1.SEM_TREE_ID AS left,
           AUTH_USER_MD5.USER_ID AS right
    FROM   SEM_TREE st1,
           SEM_TREE st2,
           SEM_TREE st3,
           SEMINAR_SEM_TREE,
           SEMINARE,
           SEMINAR_USER,
```

```
            AUTH_USER_MD5
    WHERE   st1.SEM_TREE_ID = st2.PARENT_ID
    AND     st2.SEM_TREE_ID = st3.PARENT_ID
    AND     st3.SEM_TREE_ID = SEMINAR_SEM_TREE.SEM_TREE_ID
    AND     SEMINAR_SEM_TREE.SEMINAR_ID = SEMINARE.SEMINAR_ID
    AND     SEMINARE.SEMINAR_ID = SEMINAR_USER.SEMINAR_ID
    AND     SEMINAR_USER.USER_ID = AUTH_USER_MD5.USER_ID )
DATA INITIALLY DEFERRED
REFRESH DEFERRED;

-- view4
CREATE TABLE view4
AS (SELECT SEM_TREE.SEM_TREE_ID AS left,
            SEMINARE.SEMINAR_ID AS right
    FROM    SEM_TREE,
            SEMINAR_SEM_TREE,
            SEMINARE
    WHERE   SEM_TREE.SEM_TREE_ID = SEMINAR_SEM_TREE.SEM_TREE_ID
    AND     SEMINAR_SEM_TREE.SEMINAR_ID = SEMINARE.SEMINAR_ID

    UNION

    SELECT st1.SEM_TREE_ID AS left,
            SEMINARE.SEMINAR_ID AS right
    FROM    SEM_TREE st1,
            SEM_TREE st2,
            SEMINAR_SEM_TREE,
            SEMINARE
    WHERE   st1.SEM_TREE_ID = st2.PARENT_ID
    AND     st2.SEM_TREE_ID = SEMINAR_SEM_TREE.SEM_TREE_ID
    AND     SEMINAR_SEM_TREE.SEMINAR_ID = SEMINARE.SEMINAR_ID

    UNION

    SELECT st1.SEM_TREE_ID AS left,
            SEMINARE.SEMINAR_ID AS right
    FROM    SEM_TREE st1,
            SEM_TREE st2,
            SEM_TREE st3,
            SEMINAR_SEM_TREE,
            SEMINARE,
    WHERE   st1.SEM_TREE_ID = st2.PARENT_ID
    AND     st2.SEM_TREE_ID = st3.PARENT_ID
    AND     st3.SEM_TREE_ID = SEMINAR_SEM_TREE.SEM_TREE_ID
    AND     SEMINAR_SEM_TREE.SEMINAR_ID = SEMINARE.SEMINAR_ID )
DATA INITIALLY DEFERRED
REFRESH DEFERRED;

-- view5
CREATE TABLE view5
AS (SELECT st1.SEM_TREE_ID AS left,
            st2.SEM_TREE_ID AS right
    FROM    SEM_TREE st1,
            SEM_TREE st2,
    WHERE   st1.SEM_TREE_ID = st2.PARENT_ID
```

```
    UNION

    SELECT st1.SEM_TREE_ID AS left,
           st3.SEM_TREE_ID AS right
    FROM   SEM_TREE st1,
           SEM_TREE st2,
           SEM_TREE st3,
    WHERE  st1.SEM_TREE_ID = st2.PARENT_ID
    AND    st2.SEM_TREE_ID = st3.PARENT_ID)
DATA INITIALLY DEFERRED
REFRESH DEFERRED;


---
-- Indices to speed up view access
---
CREATE Index index1view ON view1 (left);
CREATE Index index2view ON view2 (left);
CREATE Index index3view ON view3 (left);
CREATE Index index4view ON view4 (left);
CREATE Index index5view ON view5 (left);
```

Listing 10.3: Materialized view definition resulting from sample model

```
---
-- Triggers to refresh views
---

--------
-- view1
--------
CREATE TRIGGER IT_V1_seminar
AFTER INSERT ON SEMINARE
FOR EACH STATEMENT
  CALL refresher ('view1');

CREATE TRIGGER UT_V1_seminar_user
AFTER UPDATE ON SEMINAR_USER
FOR EACH STATEMENT
  CALL refresher ('view1');

CREATE TRIGGER IT_V1_seminar_user
AFTER INSERT ON SEMINAR_USER
FOR EACH STATEMENT
  CALL refresher ('view1');

--------
-- view2
--------
CREATE TRIGGER IT_V2_seminar
AFTER INSERT ON SEMINARE
FOR EACH STATEMENT
  CALL refresher ('view2');
```

```
CREATE TRIGGER UT_V2_seminar_user
AFTER UPDATE ON SEMINAR_USER
FOR EACH STATEMENT
  CALL refresher ('view2');

CREATE TRIGGER IT_V2_seminar_user
AFTER INSERT ON SEMINAR_USER
FOR EACH STATEMENT
  CALL refresher ('view2');

--------
-- view3
--------
CREATE TRIGGER UT_V3_sem_tree
AFTER UPDATE ON SEM_TREE
FOR EACH STATEMENT
  CALL refresher ('view3');

CREATE TRIGGER IT_V3_sem_tree
AFTER INSERT ON SEM_TREE
FOR EACH STATEMENT
  CALL refresher ('view3');

CREATE TRIGGER UT_V3_seminar_sem_tree
AFTER UPDATE ON SEMINAR_SEM_TREE
FOR EACH STATEMENT
  CALL refresher ('view3');

CREATE TRIGGER IT_V3_seminar_sem_tree
AFTER INSERT ON SEMINAR_SEM_TREE
FOR EACH STATEMENT
  CALL refresher ('view3');

CREATE TRIGGER UT_V3_seminar_user
AFTER UPDATE ON SEMINAR_USER
FOR EACH STATEMENT
  CALL refresher ('view3');

CREATE TRIGGER IT_V3_seminar_user
AFTER INSERT ON SEMINAR_USER
FOR EACH STATEMENT
  CALL refresher ('view3');


--------
-- view4
--------
CREATE TRIGGER UT_V4_sem_tree
AFTER UPDATE ON SEM_TREE
FOR EACH STATEMENT
  CALL refresher ('view4');

CREATE TRIGGER IT_V4_sem_tree
AFTER INSERT ON SEM_TREE
```

```
FOR EACH STATEMENT
  CALL refresher ('view4');

CREATE TRIGGER UT_V4_seminar_sem_tree
AFTER UPDATE ON SEMINAR_SEM_TREE
FOR EACH STATEMENT
  CALL refresher ('view4');

CREATE TRIGGER IT_V4_seminar_sem_tree
AFTER INSERT ON SEMINAR_SEM_TREE
FOR EACH STATEMENT
  CALL refresher ('view4');

--------
-- view5
--------
CREATE TRIGGER UT_V5_sem_tree
AFTER UPDATE ON SEM_TREE
FOR EACH STATEMENT
  CALL refresher ('view5');

CREATE TRIGGER IT_V5_sem_tree
AFTER INSERT ON SEM_TREE
FOR EACH STATEMENT
  CALL refresher ('view5');
```

Listing 10.4: Triggers to refresh materialized views

```
---
-- Triggers for actual update monitoring
---

---
-- implicit subscriptions
---

---
-- overlay1
---
CREATE TRIGGER  implicitSimple1U
AFTER UPDATE of name, ort, start_time
ON              SEMINARE
REFERENCING NEW AS new
FOR EACH ROW
BEGIN ATOMIC
INSERT INTO NOTIFICATIONS (idOfUpdatedObject,
                           typeOfUpdatedObject,
                           idOfSubscriber,
                           typeOfSubscriber)
            SELECT         new.seminar_id,
                           'SEMINARE',
                           view1.right,
                           'AUTH_USER_MD5'
            FROM           view1
```

```
            WHERE          view1.left = new.SEMINAR_ID;
END;

CREATE TRIGGER implicitSimple1I
AFTER INSERT
ON             SEMINARE
REFERENCING NEW AS new
-- remainder equal to implicitSimple1U
---
-- overlay2
---
CREATE TRIGGER  implicitSimple2U
AFTER UPDATE OF description, filename
ON              DOKUMENTE
REFERENCING NEW AS new
FOR EACH ROW
BEGIN ATOMIC
INSERT INTO NOTIFICATIONS (idOfUpdatedObject,
                           typeOfUpdatedObject,
                           idOfSubscriber,
                           typeOfSubscriber)
            SELECT         new.dokument_id,
                           'DOKUMENTE',
                           view2.right,
                           'AUTH_USER_MD5'
            FROM           view2
            WHERE          view2.left = new.DOKUMENT_ID;
END;

CREATE TRIGGER  implicitSimple2I
AFTER INSERT
ON              DOKUMENTE
REFERENCING NEW AS new
-- remainder equal to implicitSimple2U


---
-- overlay3
---
CREATE TRIGGER  implicitSimple3U
AFTER UPDATE OF info, name
ON              SEM_TREE
REFERENCING NEW AS new
FOR EACH ROW
BEGIN ATOMIC
INSERT INTO NOTIFICATIONS (idOfUpdatedObject,
                           typeOfUpdatedObject,
                           idOfSubscriber,
                           typeOfSubscriber)
            SELECT         new.SEM_TREE_ID,
                           'SEM_TREE',
                           view3.right,
                           'AUTH_USER_MD5'
            FROM           view3
            WHERE          view3.left = new.SEM_TREE_ID;
```

```
END;

CREATE TRIGGER implicitSimple3I
AFTER INSERT
ON              SEM_TREE
REFERENCING NEW AS new
-- remainder equal to implicitSimple3U


---
-- explicit subscriptions
---

---
-- overlay1
---
CREATE TRIGGER  explicitSimple1U
AFTER UPDATE of name, ort, start_time
ON              SEMINARE
REFERENCING NEW AS new
FOR EACH ROW
BEGIN ATOMIC
INSERT INTO NOTIFICATIONS (idOfUpdatedObject,
                           typeOfUpdatedObject,
                           idOfSubscriber,
                           typeOfSubscriber)
            SELECT         new.seminar_id,
                           'SEMINARE',
                           es.idOfSubscriber,
                           es.typeOfSubscriber
            FROM           ExplicitSubscription es
            WHERE          es.idOfSubscribable = new.SEMINAR_ID
            AND            es.typeOfSubscribable = 'SEMINARE';
END;

CREATE TRIGGER  explicitSimple1I
AFTER INSERT
ON              SEMINARE
REFERENCING NEW AS new
-- remainder equal to explicitSimple1U

---
-- overlay2
---
CREATE TRIGGER  explicitSimple2U
AFTER UPDATE OF description, filename
ON              DOKUMENTE
REFERENCING NEW AS new
FOR EACH ROW
BEGIN ATOMIC
INSERT INTO NOTIFICATIONS (idOfUpdatedObject,
                           typeOfUpdatedObject,
                           idOfSubscriber,
                           typeOfSubscriber)
            SELECT         new.dokument_id,
```

```
                            'DOKUMENTE',
                            es.idOfSubscriber,
                            es.typeOfSubscriber
            FROM            ExplicitSubscription es
            WHERE           es.idOfSubscribable = new.DOKUMENT_ID
            AND             es.typeOfSubscribable = 'DOKUMENTE'
            UNION
            SELECT          new.dokument_id,
                            'DOKUMENTE',
                            es.idOfSubscriber,
                            es.typeOfSubscriber
            FROM            ExplicitSubscription es,
                            seminare
            WHERE           new.SEMINAR_ID = seminare.SEMINAR_ID
            AND             es.idOfSubscribable = new.SEMINAR_ID
            AND             es.typeOfSubscribable = 'SEMINARE';
END;

CREATE TRIGGER  explicitSimple2I
AFTER INSERT
ON              DOKUMENTE
REFERENCING NEW AS new
-- remainder equal to explicitSimple2U

---
-- overlay3
---
CREATE TRIGGER  explicitSimple3U
AFTER UPDATE OF info, name
ON              SEM_TREE
REFERENCING NEW AS new
FOR EACH ROW
BEGIN ATOMIC
INSERT INTO NOTIFICATIONS (idOfUpdatedObject,
                            typeOfUpdatedObject,
                            idOfSubscriber,
                            typeOfSubscriber)
            SELECT          new.SEM_TREE_ID,
                            'SEM_TREE',
                            es.idOfSubscriber,
                            es.typeOfSubscriber
            FROM            ExplicitSubscription es
            WHERE           es.idOfSubscribable = new.SEM_TREE_ID
            AND             es.typeOfSubscribable = 'SEM_TREE'
            UNION
            SELECT          new.SEM_TREE_ID,
                            'SEM_TREE',
                            es.idOfSubscriber,
                            es.typeOfSubscriber
            FROM            ExplicitSubscription es,
                            view4
            WHERE           view4.left = new.SEM_TREE_ID
            AND             es.idOfSubscribable = view4.right
            AND             es.typeOfSubscribable = 'SEMINARE';
            UNION
```

```
            SELECT          new.SEM_TREE_ID,
                            'SEM_TREE',
                            es.idOfSubscriber,
                            es.typeOfSubscriber
            FROM            ExplicitSubscription es,
                            view5
            WHERE           view5.left = new.SEM_TREE_ID
            AND             es.idOfSubscribable = view5.right
            AND             es.typeOfSubscribable = 'SEM_TREE';

END;

CREATE TRIGGER  explicitSimple3I
AFTER INSERT
ON              SEM_TREE
REFERENCING NEW AS new
-- remainder equal to explicitSimple3U
```

Listing 10.5: Trigger definition resulting from sample model

10.7 Summary

In this chapter, we finally presented the UML profile representing all proposed event-handling constructs as well as a brief description of the transformations, transforming models (enriched with elements from the UML profile) into event-handling database triggers. Giving this information, together with a proof that the transformations work correctly, the model-driven realization of our concept has been described. Thus, our model-driven implementation of the non-invasive event-handling approach can now be evaluated qualitatively and quantitatively, which is subject to the remaining last part of this dissertation.

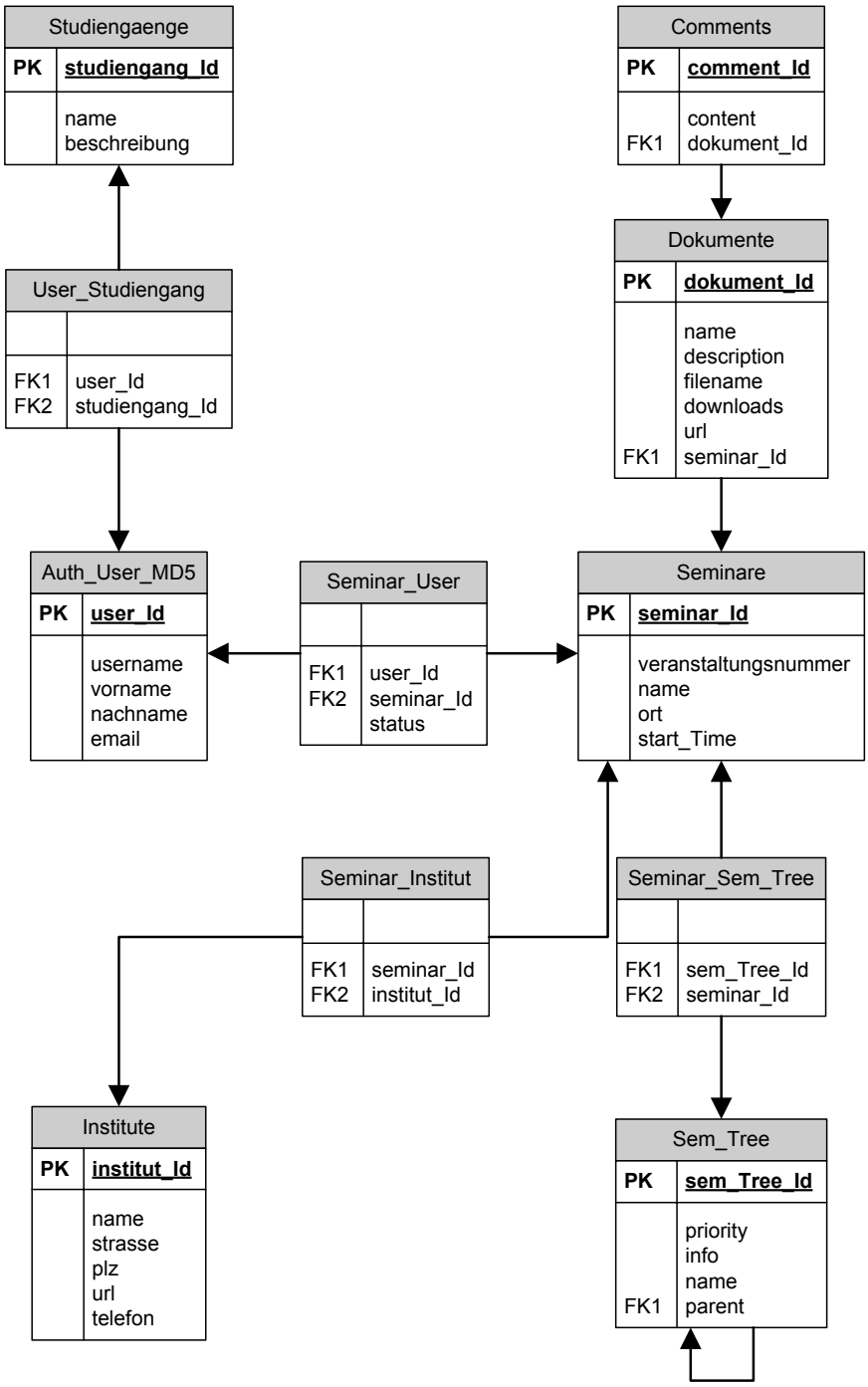


Figure 10.6: Entity-relationship model for Stud.IP's object-oriented data model

Part IV

Résumé

"Experience is that marvelous thing that enables you to recognize a mistake when you make it again."

Franklin P. Jones

11 Critical Evaluation

As for any good research project, finding and implementing a solution to a given problem is only half the battle: equally important is an objective evaluation and a presentation of the lessons learnt. In this chapter, we will therefore evaluate our proposed solution using objective criteria. To do so, we divide the evaluation into three parts, ordered in increasing level of abstraction: first, we rate our prototypic implementation based on Model Driven Architecture, database triggers and materialized views. Next, our event-handling concept and comprehensive architectural proposal is evaluated. Finally, on the highest level of abstraction, we rate the usage of a generative approach in general. In addition to objectively evaluating our approach against all relevant requirements and criteria, we compare various elements of our solution with related technologies and approaches, where appropriate. Last, and most important, we present our insights about the circumstances and conditions under which the usage of generative approaches, such as MDA, should be encouraged.

11.1 Evaluation of the Prototypic Implementation

The evaluation of the prototypic active database implementation with MDA is divided into two parts: first, we apply the objective criteria of the international standard

for software quality *ISO 9126* [Wika]. Second, experiments measuring the actual performance and scalability of the generated database triggers were conducted. The results of these experiments are presented in the second part of this section.

11.1.1 Software Quality According to ISO 9126

The international standard ISO 9216 [Wika] defines a set of requirements that should be used to evaluate software quality. These requirements are divided into six categories, addressing different aspects of software quality, each containing several characteristics. Although ISO 9126 has been replaced by the more comprehensive standard ISO/IEC 25000 since 2005, its quality model can still act as a guide to evaluate software quality.

Thus, in the following, our MDA solution is briefly checked against the requirements of ISO 9126.

11.1.1.1 Functionality

This category contains aspects that bear on the functions realized within the software system to be evaluated and comprises several requirements.

First of all, there is the question of **Suitability**: Does the software satisfy the functional requirements? In our case, this question rather applies to the event-handling concept than to the MDA implementation. However, as we will show in section 11.2, the concept itself fulfils all of our functional requirements for event-handling system. Thus, the implementation (which has been proven to correctly implement the concept) also satisfies this requirement.

Furthermore, **Accuracy** has to be taken into account: Does the software deliver correct results? Since we consider the results of our MDA solution to be correct if it delivers results that cohere to our formal concept and since correctness has been proven in chapter 10, this requirement is fulfilled, too.

The next requirement concerns **Interoperability**: Does the software interoperate with legacy systems? In our context, interoperability can mainly be defined as the possibility to integrate our concept with legacy systems. Since this is mainly an architectural/conceptual requirement, we will show in section 11.2 why this requirement can be considered as satisfied.

A requirement which might be important in other fields of application is the question of **Compliance**: Does the software comply to the appropriate legal and/or application-specific standards? This requirement is not applicable to our scenario and thus has not been evaluated.

Finally, **Security** issues must be considered: Does the software restrict unauthorized access to its data? The aspect of security is an important matter in our scenario: by allowing users to subscribe to sensitive data and/or sending them notifications about updated data, existing security mechanisms can possibly be undermined. However, security mechanisms were not in the focus of our work, so we have to admit that this requirement is not fulfilled. However, this topic has been identified as an open issue and is listed as an open end in chapter 12. Thus, it should be treated in the course of future research.

11.1.1.2 Reliability

In this category, the ability of the software to maintain its level of performance for a stated period of time is considered. This is expressed by the requirements **Maturity**, **Recoverability** and **Fault Tolerance**. Since our solution has been developed as a prototype to prove the adequacy of our concept, no effort has been put into matters of reliability.

11.1.1.3 Usability

Aspects of usability are also important criteria for the quality of software. ISO 9126 mentions three very similar requirements: can users **learn**, **understand** and **operate** the software system with minimal effort? In our case, these requiremens have to be evaluated with respect to two different user groups: information system designers and users of the respective system.

Users of the generated system have a very small and very simple interface to the event-handling functionality: they can explicitly subscribe to particular entities and they get notifications about updates, containing information about the updated bit of information. Being this the only two functionalities the user comes in touch with, it is rather easy to learn, understand and operate the system. The interesting parts of our software, i.e. the implicit notifications and event-propagations, remain invisible to regular users and thus need no effort from the users.

Designers of the event-handling system, however, have to learn and understand the different constructs of our concept and the underlying event-handling semantics. However, the required effort is minimal: on the one hand, standardized, well-known technologies, such as UML and MDA have been used, minimizing the need to become acquainted with new technologies. On the other hand, only a handful of constructs have been incorporated into our concept, so that both the initial effort to learn as well as the effort to operate our framework are minimal.

11.1.1.4 Efficiency

The aspect of efficiency comprises the two domains **Time Behaviour** and **Resource Behaviour**. Since the efficiency of our approach is of central importance in real-life information systems, we will take a closer look at the runtime performance and scalability of our generated database triggers in section 11.1.2.

11.1.1.5 Maintainability and Portability

Since the two aspects maintainability and portability are closely related to each other, we evaluate them together. Their most important requirements are **Changeability** and **Adaptability**: the easier a software system can be adopted to new functional and non-functional requirements and/or (technical) platforms, the better it is. This is where our approach profits from its universality. As we already presented in chapter 4, several layers of abstraction allow the easy adaption of our framework to different needs. For instance, instead of generating triggers for relational databases, monitors for the observation of text files could be generated, using the same semantic declarations. Similarly, object-oriented databases or XML databases could be the target platform for our solution.

Furthermore, maintainability and extendability concerning new functional requirements is broadly supported: as soon as a new event-handling construct has to be incorporated into our framework, this is easily possible by adding those new constructs to the formal model, defining clear semantics of how to interpret them, add them to the UML profile and develop appropriate transformations to generate the respective code fragments afterwards. Although this might sound easier than it actually is, anticipation of change can not be supported any better than by providing the appropriate levels of abstraction and the interfaces in between, as we did.

11.1.2 A Detailed View on Performance and Scalability

To get results about the performance and scalability of our approach, we did several experiments. They can be divided into three catagories: in the first category, we evaluated the scalability of our database triggers without optimiziation. The second category evaluated how the introduction of materialized views improves the overall performance under certain boundary conditions. Finally, using a copy of the actual Stud.IP data of the University of Passau and a series of typical update statements within this application, real-life applicability and adequacy were evaluated.

11.1.2.1 Performance and Scalability in General

Now that important qualitites of our solution have been shown, we take a closer look at the performance of the generated trigger based solution. We therefore constructed a reference model containing all relevant concepts.[1] This reference model is depicted in figure 11.1 and models part of a digital library system: users can specify their interest in specific topics (realized as Terms), which themselves are related to each other via the isSubtopic association, thus forming a taxonomy. All documents can deal with several of these terms. Whenever a document is updated or created, the respective term(s) are considered as updated as well (using the event-propagating association dealsWith). Along the implicit subscription isInterested, all library users are automatically informed about relevant updates.

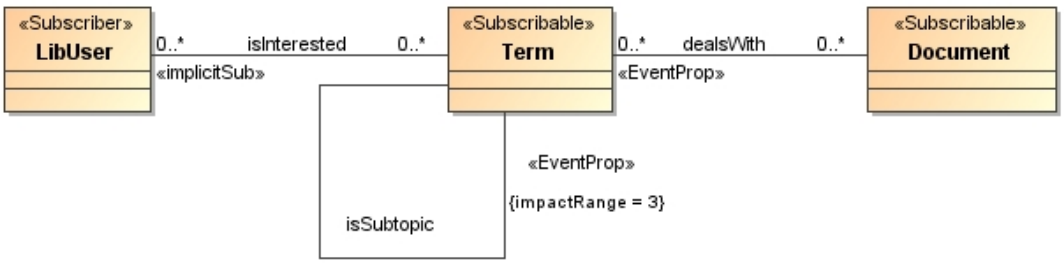


Figure 11.1: UML model used to measure the efficiency of our approach

[1] Since different overlays within one information system's model are treated independently, the size of the overall model and the number of overlays is not critical to performance. Instead, the relevant factor is the size of the database and the specification of the overlay. It is thus sufficient to use a model containing all event-handling concepts, containing a realistic number of entities.

In our test scenario, a taxonomy resembling a tree of degree *degree* and depth *depth* was built. For every term in this taxonomy, an isInterested association for each simulated user had been stored in ten percent of all cases. To measure the efficiency of our approach based on these artificial scenarios, 10 percent of all taxonomy terms were updated by random and the response time required for this update (including the trigger execution time) was measured.

Scalability was considered along three different dimensions: the number of simulated users, i.e. scalability with respect to the number of subscribers and with respect to the size of the taxonomy (regarding *depth* and *degree* of the taxonomy tree). Each of these test series were conducted using three different UML models which differed from each other only in the impactRange (1, 3 or 5) of the reflexive association isSubtopic. To get a feeling for the actual costs of the event-handling triggers, the update times were also measured without any triggers at all.

All measurements were conducted on a 2.4 GHz PC with 1GB of ram. To get representative results, all tests were run four times; the result of the first run was ignored in order to avoid any initializing effects. The average result of runs two to four was then used as the final result.

On the following pages, these results are presented and interpreted.

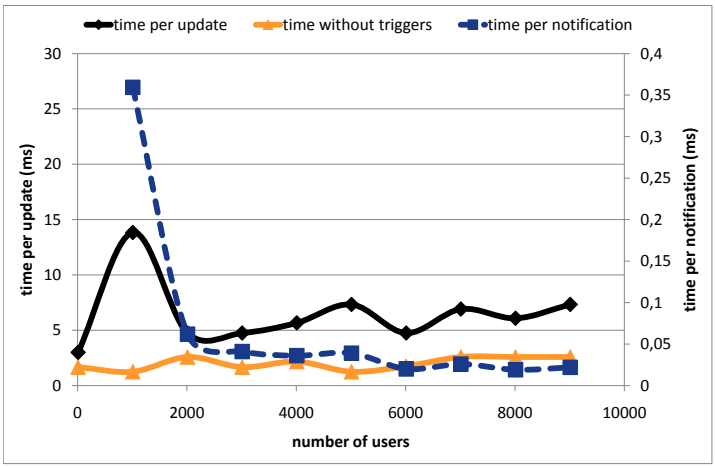
Scalability with the amount of users To determine the scalability with the amount of users, we used a taxonomy of depth 4 and degree 3. Between 10 and 10,000 users were simulated. Table 11.2(a) shows the results:

The results are presented graphically in figure 11.2(b): the curves indicate the time per update in milliseconds, the average time per update normalized by the amount of notifications that were caused, and the time per update without triggers as a reference.

As one would expect, the time per update without triggers remains constant, because only the terms are updated, i.e. the number of users does not affect the updates at all. More interestingly, the time per update levels off around 7 ms per update, so that a larger amount of users does not significantly influence the performance per update, which is a good indication for linear performance with respect to the number of subscribers. As more and more notifications have to be stored for an increased amount of users (remember that each user had a 10 percent probability to be interested in an arbitrary topic) while the time per update remains constant, the average time per update decreases which again is a good sign for the overall scalability.

	No Triggers	*Triggers*	
#users	ms/update	ms/update	ms/notification
10	20	3	n/a
1010	15	13.83	0.360
2010	31	4.75	0.062
3010	20	4.75	0.041
4010	26	5.67	0.036
5010	15	7.33	0.039
6010	21	4.75	0.020
7010	31	6.92	0.026
8010	31	6.08	0.019
9010	31	7.33	0.022

(a) Test results



(b) Graphical Representation

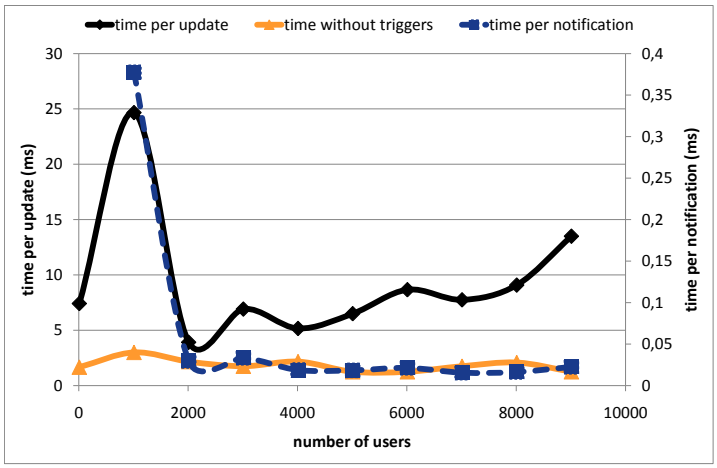
Figure 11.2: Scalability with number of users - ImpactRange 1

The same situation can be observed when applying our tests to a model with an impactRange of 3, as shown in table 11.3(a) and figure 11.3(b).

Again, the time per update slightly increases linearly while raising the number of users, thus also leading to a rather constant update time per notification. However, the required update times are slightly higher than in the previous case with impactRange 1, since a higher impactRange means that more terms are transitiveley considered to be updated, thus causing more subscribers to be notified.

	No Triggers	*Triggers*	
#users	ms/update	ms/update	ms/notification
10	20	7.42	n/a
1010	36	24.67	0.377
2010	26	3.92	0.030
3010	21	6.92	0.033
4010	26	5.17	0.019
5010	15	6.50	0.019
6010	15	8.67	0.021
7010	21	7.75	0.016
8010	25	9.08	0.016
9010	15	13.50	0.023

(a) Test results



(b) Graphical Representation

Figure 11.3: Scalability with number of users - ImpactRange 3

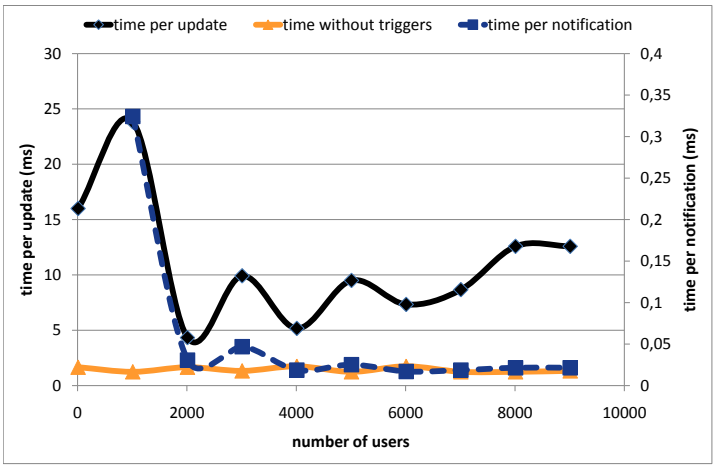
Looking at the results for the model with impactRange 5 (table 11.4(a) and figure 11.4(b)), the same tendency is visible. Even more interesting is that the average update times do not significantly increase compared to impactRange 3.

Those results give evidence to the claim that the proposed solution scales very well with the amount of subscribers - the most common case in real-life scenarios.

Another important aspect is the scalability with the number of subscribables, i.e. the size of the taxonomy. As our test-taxonomy consists of a complete n-ary tree, we can

	No Triggers	*Triggers*	
#users	ms/update	ms/update	ms/notification
10	20	16.00	n/a
1010	15	23.83	0.324
2010	20	4.33	0.031
3010	16	9.92	0.047
4010	21	5.17	0.019
5010	15	9.50	0.025
6010	21	7.33	0.017
7010	15	8.67	0.018
8010	15	12.58	0.022
9010	16	12.58	0.021

(a) Test results



(b) Graphical Representation

Figure 11.4: Scalability with number of users - ImpactRange 5

measure overall performance with respect to two dimensions: depth of the taxonomy and degree of the taxonomy.

To be able to better interpret the results, it is important to know that the overall number of nodes (i.e. terms, in our case) within the taxonomy can be expressed as

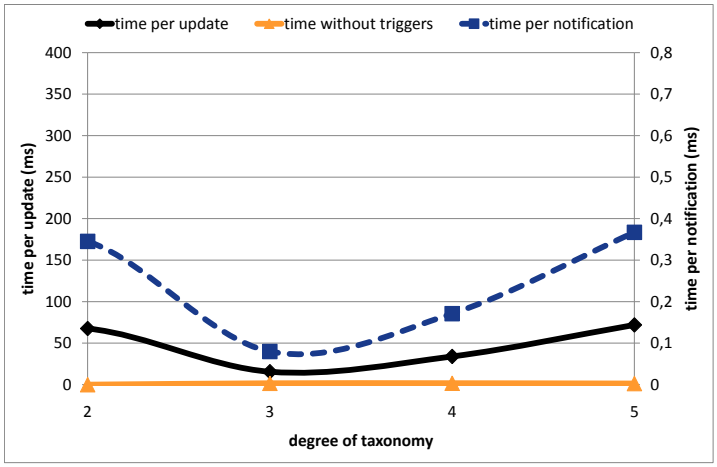
$$\sum_{i=0}^{depth} degree^i$$

Thus, a linear increase of the depth obviously leads to an exponential growth of terms whilst a linear increase of the degree leads to a linear/quadratic/cubic/... growth, depending on the depth of the taxonomy.

Scalability with the Degree of the Taxonomy We simulated a taxonomy of depth 4 and 5000 users. The degree of the taxonomy was variied between 1 and 5. Again, the experiments were conducted on three models differing in the value of the impactRange. Table 11.5(a) and figure 11.5(b) show the results for the model with impactRange 1, revealing a slight and almost linear growth of the average update times.

	No Triggers	*Triggers*	
degree	ms/update	ms/update	ms/notification
1	n/a	62.00	0.373
2	5	67.67	0.345
3	21	15.58	0.080
4	46	34.00	0.161
5	124	71.97	0.367

(a) Test results



(b) Graphical Representation

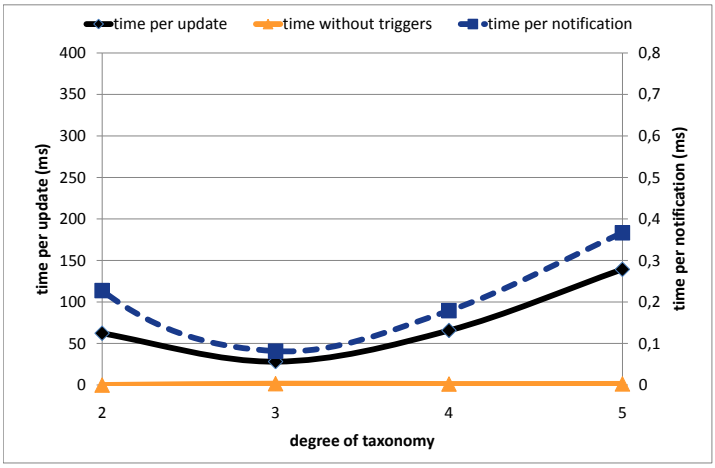
Figure 11.5: Scalability with degree of taxonomy - ImpactRange 1

The same tendency can be recognized in tables 11.6(a) and 11.7(a) and the corresponding diagrams 11.6(b) and 11.7(b) - a higher impact range obviously increases

the response times, but not the tendency of the results. This can be explained by the fact that the depth of the taxonomy was kept constant at 4: thus, increasing the impact range from 3 to 5 is not significantly influencing the effort for the determination of subscribers, because at most paths of length 4 from leaf nodes to the root node are contained in the data.

	No Triggers	*Triggers*	
degree	ms/update	ms/update	ms/notification
1	n/a	104.00	0.727
2	5	62.33	0.227
3	15	28.17	0.082
4	47	65.71	0.179
5	114	139.36	0.367

(a) Test results



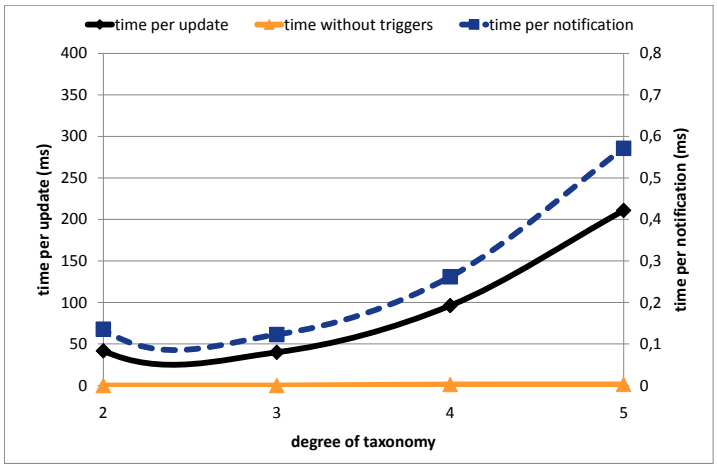
(b) Graphical Representation

Figure 11.6: Scalability with degree of taxonomy - ImpactRange 3

Scalability with the Depth of the Taxonomy Finally, for the scalability with the depth of the taxonomy, we used a taxonomy of degree 2 and 2000 users. The depth was varied between 2 and 8. The results for impactRanges of 1 to 5 are shown in figures 11.8(b) to 11.10(b) and tables 11.8(a) to 11.10(a).

	No Triggers	*Triggers*	
degree	ms/update	ms/update	ms/notification
1	n/a	229.00	0.674
2	n/a	41.67	0.135
3	15	39.92	0.122
4	46	96.19	0.262
5	109	210.81	0.571

(a) Test results



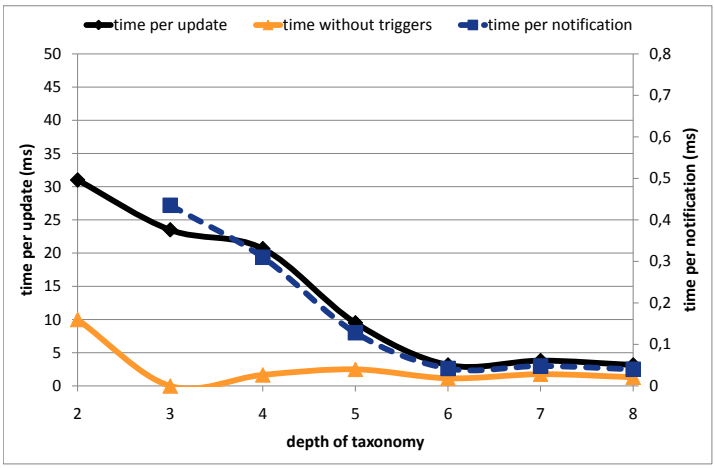
(b) Graphical Representation

Figure 11.7: Scalability with degree of taxonomy - ImpactRange 5

In all three scenarios, the average time per update decreases as the depth of the taxonomy increases. Since the impact range limits the effect of the event propagation, there is a taxonomy depth from which on the average times per update remain almost constant. As an additional result to the previous scenarios, we can thus state that the depth of a taxonomy only influences the average update times as long as the impact range, following the taxonomy from root to leaf (or vice-versa) is higher than the maximum depth of a taxonomic structure.

	No Triggers	*Triggers*	
depth	ms/update	ms/update	ms/notification
2	10	31.00	0.525
3	n/a	23.50	0.435
4	5	20.67	0.310
5	15	9.50	0.128
6	15	3.15	0.042
7	46	3.81	0.048
8	67	3.16	0.040

(a) Test results



(b) Graphical Representation

Figure 11.8: Scalability with depth of taxonomy - ImpactRange 1

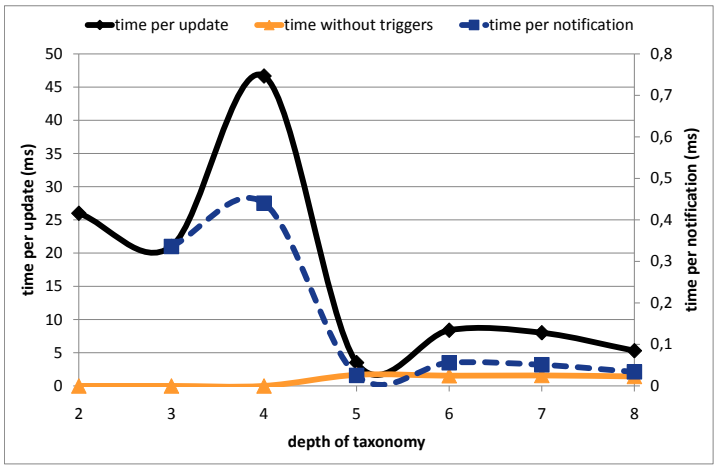
11.1.2.2 Benefit of Optimization

To evaluate the benefit of the optimization approach using materialized views as event propagation indices, a series of additional tests were run. The test setup consisted of the same data model that was used for the previous experiments, however, an event propagation index view had been created, storing all paths from documents along the taxonomy to the implicit subscribers. An impact range of 5 was used throughout all of the following experiments.

Furthermore, in our test scenario, the updates *only* updated documents but did not update any of the taxonomic associations *isSubtopic*. Thus, the event propagation

	No Triggers	*Triggers*	
depth	ms/update	ms/update	ms/notification
2	n/a	26.00	0.456
3	n/a	21.00	0.336
4	n/a	46.67	0.440
5	10	3.50	0.025
6	20	8.38	0.056
7	41	8.00	0.051
8	73	5.29	0.034

(a) Test results



(b) Graphical Representation

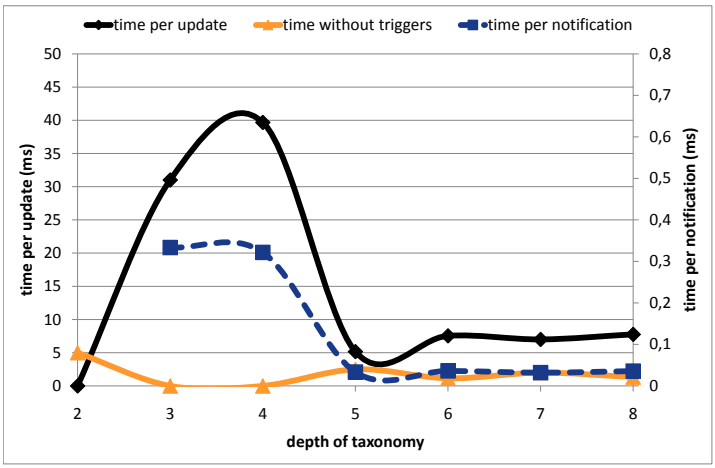
Figure 11.9: Scalability with depth of taxonomy - ImpactRange 3

index did not have to be updated in any of the cases, so that we were able to show the maximum potential of the optimization approach. In real-life use cases, however, the actual benefit can be expected to be less significant.

Optimization With Respect to Taxonomy Depth To evaluate the tendency of the optimization benefit with an increasing taxonomy depth, our scenario was evaluated with a taxonomy of degree 3 and depth between 2 and 5, simulating a total of 10,000 users. The results of these experiments are shown in table 11.11(a) and figure 11.11(b).

	No Triggers	*Triggers*	
depth	ms/update	ms/update	ms/notification
2	5	n/a	n/a
3	n/a	31.00	0.333
4	n/a	39.67	0.322
5	15	5.17	0.033
6	15	7.54	0.036
7	52	7.00	0.032
8	68	7.76	0.036

(a) Test results



(b) Graphical Representation

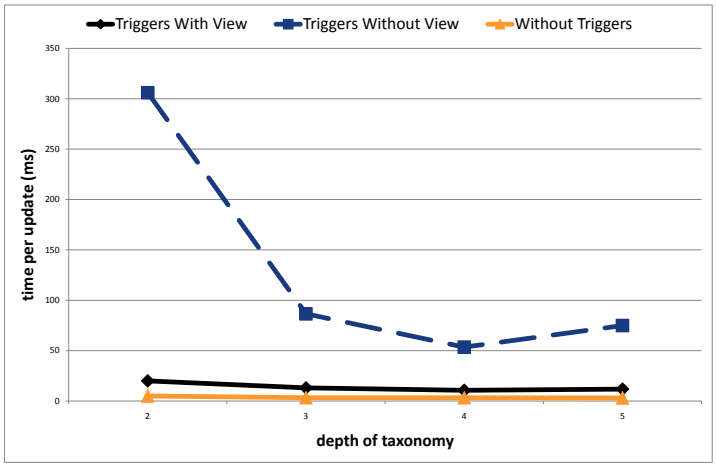
Figure 11.10: Scalability with depth of taxonomy - ImpactRange 5

As we can see, the optimized and the non-optimized scenario show the same scalability tendency. However, the optimized approach leads to an improvement factor of about 20, i.e. the optimized queries ran up to 20 times faster than their non-optimized alternatives. However, the notification functionality still has to be paid by update queries that are about 4 times slower than the updates without any optimization.

Optimization With Respect to Taxonomy Degree Similar results were obtained during the tests evaluating the scalability with the degree of the taxonomy. Using a constant depth of 3 and degrees between 2 and 5 (again with 10,000 users), the optimized approach again leads to queries about 20 times faster, while the optimized

depth	ms/update w/ views	ms/update w/o views	ms/update w/o triggers
2	20.0	306.0	5
3	13.0	86.5	3.3
4	10.6	53.5	3.3
5	11.8	74.9	3.0

(a) Test results



(b) Graphical Representation

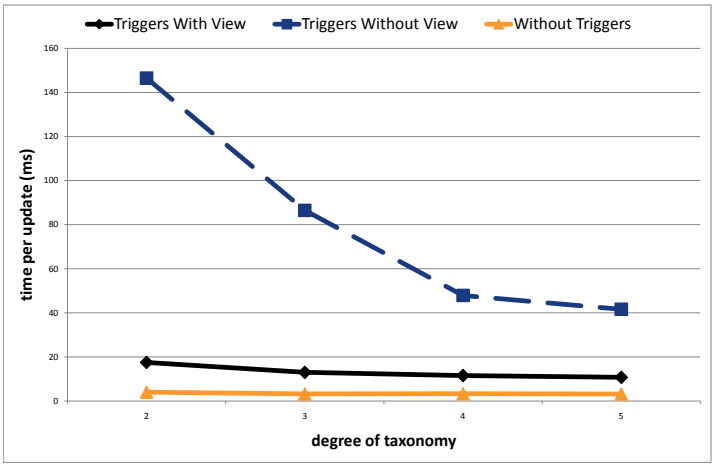
Figure 11.11: Benefit of optimization (Dimension: depth of taxonomy)

approach still is 3 to 4 times slower than updates without any notification functionality, as shown in table 11.12(a) and figure 11.12(b).

Optimization With Respect to User Count As a final scenario, the optimization impact regarding a growing number of simulated users was evaluated. Using a taxonomy of depth 3 and degree 3, up to 20,000 users were simulated and the responding update times were recorded, shown in table 11.13(a) and figure 11.13(b). Like in the previous two cases, we observe similar linear scaling in all three cases, again yielding an optimization factor of about 20 between optimized and non-optimized triggers.

degree	ms/update w/ views	ms/update w/o views	ms/update w/o triggers
2	17.5	146.5	4
3	13.0	86.5	3.3
4	11.6	47.9	3.3
5	10.8	41.6	3.2

(a) Test results



(b) Graphical Representation

Figure 11.12: Benefit of optimization (Dimension: degree of taxonomy)

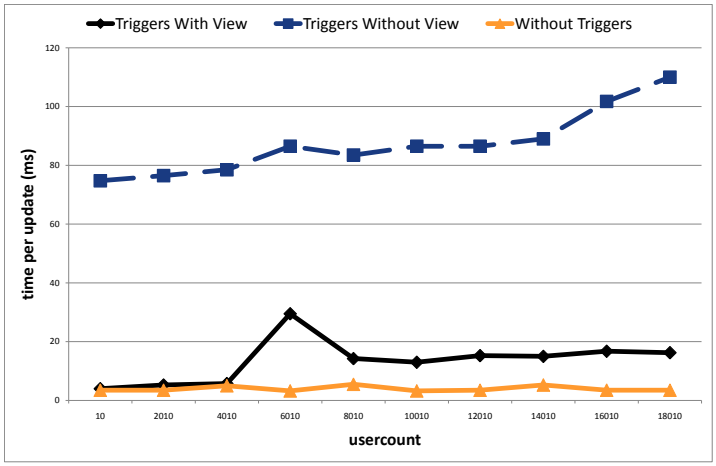
11.1.2.3 Real-Life Scenario

For a final evaluation of the practical applicability of our prototypic implementation, a copy of the actual Stud.IP data was transferred into a DB2 database and the three different overlays from figure 10.5 were applied to the database. Using this test setup, four different kinds of typical updates were executed and the average response times were recorded.

The different updates can be classified into two categories: on the one side, updates that potentially lead to notifications but do not modify any of the materialized views (i.e. the event propagation indices) were issued; on the other side, updates that lead to a rebuild of the materialized views were evaluated, too. Since the actual response times (in milliseconds) are strongly dependent on the used hardware, we also issued

# users	ms/update w/ views	ms/update w/o views	ms/update w/o triggers
10	4.0	74.8	3.5
2010	5.3	76.5	3.5
4010	5.8	78.5	5.0
6010	29.5	86.5	3.3
8010	14.3	83.5	3.5
10010	13.0	86.5	3.3
12010	15.3	86.5	3.5
14010	15.0	89.0	5.3
16010	16.8	101.8	3.5
18010	16.3	110.0	3.5

(a) Test results



(b) Graphical Representation

Figure 11.13: Benefit of optimization (Dimension: user count)

the same updates without any triggers, so that the results can be compared to each other.

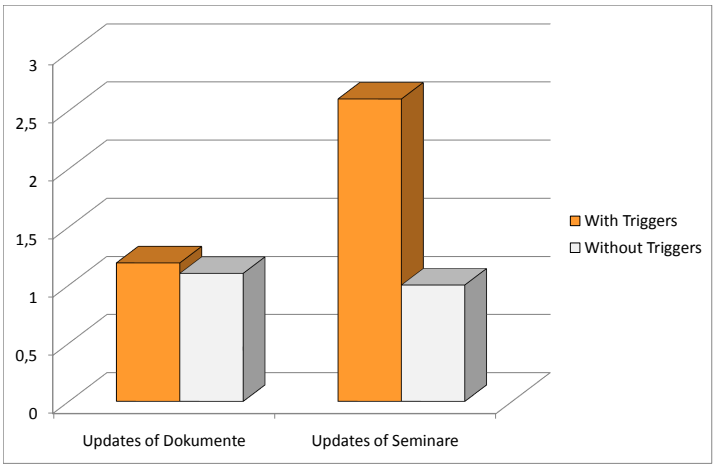
Updates Leading to Notifications To determine the impact of our notification triggers to the response time of subscriber determination, two kinds of typical updates were evaluated:

- Random elements of table Dokumente were updated, so that the respective overlay implementation leads to a number of notifications (in average, 357 notifications were created during our test runs).
- Random elements of table Seminare were updated too, also leading to a set of notifications (214 in average).

The results of the evaluation can be found in figure 11.1.2.3.

	No Triggers	*Triggers*
updated entity	ms/update	ms/update
Dokumente	1.1	1.2
Seminare	1.0	2.6

(a) Test results



(b) Graphical Representation

Figure 11.14: Performance of updates leading to notifications only

As we can learn from this evaluation, the results confirm the tendencies from the previous experiments: as long as no views are affected by the updates, the notification functionality leads to a two to three times worse performance than without any notification functionality. The different update times for Dokumente and Seminare can further be explained by the different number of notifications that were caused by the respective updates.

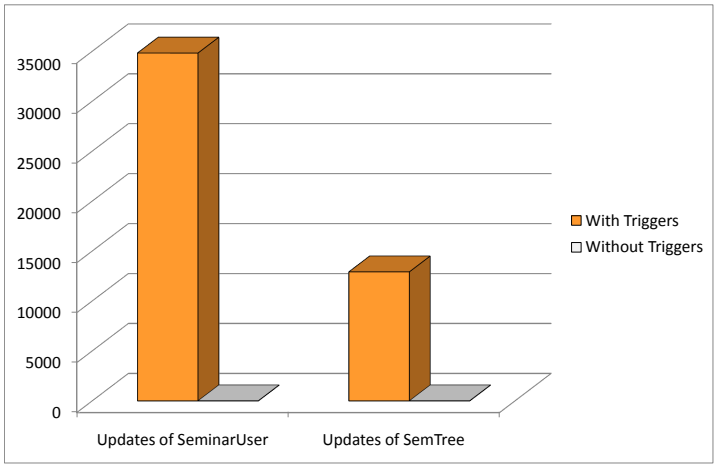
Updates Leading to Materialized View Updates In contrast, two scenarios where updates do not only trigger notifications but also lead to a recomputation of the materialized views were evaluated, too:

- Random elements of table SeminarUser were inserted, leading to the need to recompute materialized views.
- Random elements of table SemTree were inserted too, also causing view refresh triggers to be fired plus an average of 1,238 notifications.

The results, shown in figure 11.1.2.3, reveal an unacceptable decrease of performance - updates take up to 10,000 times longer than without any triggers.

	No Triggers	*Triggers*
updated entity	ms/update	ms/update
SeminarUser	3.5	34,820.9
SemTree	3.3	12,909.6

(a) Test results



(b) Graphical Representation

Figure 11.15: Performance of updates leading to notifications and view updates

However, we assume that this performance flaw is caused by the fact that our view maintenance algorithm has been realized in a simple, but ineffective way: the whole materialized view is recomputed whenever one of the underlying entities is updated,

leading to a large and - in most cases - unnecessary maintenance overhead. To improve this behaviour, it would be necessary to implement a more sophisticated index maintenance algorithm which is able to recompute only those parts of the materialized view which actually have to be changed due to the underlying update. However, we did not realize such a strategy in our prototypic implementation, but refer to this issue as an open end in chapter 12. Furthermore, we know that updates leading to view recomputations occur rather seldomly, since the underlying tables which constitute the basis of the view are updated seldomly themselves - which is the reason why they were chosen by the optimizer as candidates for precomputations. Thus, we consider this performance an issue that has to be handled, but not a significant deficit of our overall approach in general.

11.1.2.4 Overall Performance Results

Summing up all previously presented results, we can state that our approach scales properly (i.e. linear) with respect to the most important scalability factors in real-life use cases: the number of users, the depth and the degree of taxonomic structures, which can often be observed in information systems. We can also observe that the optimization in fact leads to a significant performance boost, at least if the data underlying the event propagation indices remain constant, so that the indices do not have to be updated (a theoretical evaluation and its verification of the different update scenarios has already been presented in chapter 7).

However, we must admit that the event-handling functionality, as presented in our prototypic implemention, leads to updates which are about 3 to 4 times slower than without the notification triggers. As a result, we propose to implement and evaluate an offline notification system decoupling the actual updates and the determination of subscribers, so that the updates of in-business data are not slowed down, since the determination of subscribers takes place asynchronously. Therefore, we take this proposal as a starting point for future research and list it as an open end in chapter 12.

11.1.3 Applicability in Distributed Environments

Large information systems are almost always part of a distributed software landscape. On the presentation layer, users at different locations and possibly using different clients access the information system, whilst on the storage layer, distributed database systems, possibly even with different schemata, may each contain separate parts of

the overall data pool, as figure 11.16 illustrates. Thus, the possibility of an adequate monitoring of changes in such a distributed environment can significantly improve the usefulness of a distributed information system.

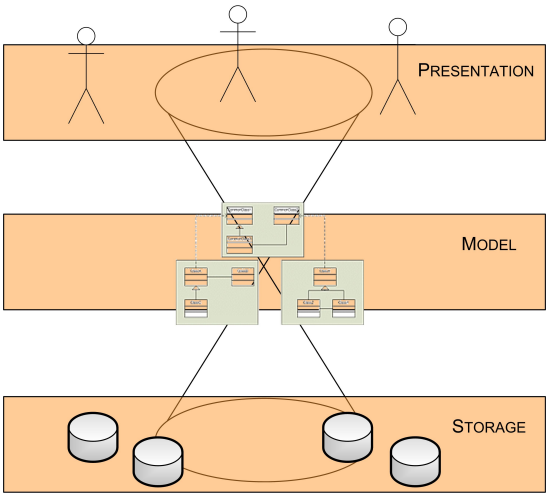


Figure 11.16: Applicability in distributed environments

Although not initially designed for such scenarios, our approach also supports this requirement: the presentation layer is out of the focus of our work; however, standard technologies such as web applications or web portals easily allow distributed access to the event-handling system.

Existing technologies can also be used to support distribution on the storage layer. Protocols like two-phase-commit guaranteeing transactional access to distributed databases are well-known and can easily be integrated into our approach, so that access to the databases is handled consistently.

What remains to evaluate is how to handle the different data models and schemata of the single storage instances. This can for instance be done as presented in figure 11.17: in the field of data integration [SPD92, BKLW99, Con02], a common integrated data model has to be designed and the individual models have to be mapped onto this common data model. To realize this mapping, we developed a solution using MDA [KGF06].

Since the application for distributed databases was not in the focus of our work, we can only give a brief solution outline for this scenario: to use our event-handling approach in a distributed, integrative environment, it is necessary to specify the event-handling semantics upon the integrated schema. Following the mapping between integrated

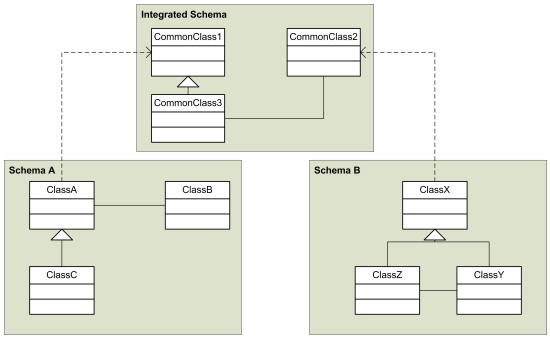


Figure 11.17: Integration of heterogeneous data models

schema and legacy schemata, it is then possible to generate an integrative view onto the legacy databases; furthermore, the necessary triggers to monitor updates within the legacy databases can be derived from the event-handling specification in combination with the mapping information.

To actually realize such an integrative event-handling system, further research is definitively necessary; however, from an abstract view we can state that our approach is theoretically applicable to distributed information systems; an issue that will be listed as an open end of our work in chapter 12.

11.1.4 Technological Alternatives

If technological alternatives to our prototypic implementation shall be evaluated, they have to be compatible (and thus comparable) to the technologies we chose. This means that we have to evaluate alternative code-generating approaches which support the event-handling concept we developed and the target architecture we proposed. We therefore evalute alternatives to MDA, database triggers and materialized views.

11.1.4.1 Alternatives to MDA

Model Driven Architecture has been developed by the OMG as a standardized procedure for the model driven software development (MDSD). However, different ways to implement MDSD are possible: proprietary solutions using custom-developed meta-meta-models and domain specific languages, proprietary code generators and proprietary template languages are actually in use in many companies.

At first appearance, it seems that the usage of MDA has no advantage over proprietary solutions: the domain specific language representing our event-handling concept can be designed using any arbitary formalism and the code generating templates can also be implemented gratuitously. *Mokum* [vdR08], a more than 25 year old approach to design information systems and generate implementations from it, would be an example for such a development framework which could be used similar to MDA. In addition, the significant difference between MDA and proprietary MDSD approaches is not used in our solution: instead of transforming the platform independent model (PIM) into a platform specific model (PSM) and *then* to code, we skip the PSM and transform *directly* to code.

However, using MDA and thus working with UML models and profiles offers a series of advantages that cannot be disregarded: UML and UML profiles, being state-of-the-art standards, are supported by a variety of modelling tools. This means that no effort is necessary to develop editors for our event-handling DSL. Another advantage of MDA is that most legacy information systems' models are available as UML models, so that our event-handling constructs can be built directly on top of those UML models without any technological disruption. Finally, as we already showed, the MDA community already developed a lot of support to describe the object-relational mapping from UML models to database languages, so this is another important advantage that one can make use of if MDA is used instead of an arbitrary code-generating solution.

11.1.4.2 Alternatives to Database Triggers

If we take a closer look at the potential alternatives to database triggers, there is hardly a technology or programming paradigm that could be used instead. If one would accept to switch from the "active push" paradigm to a polling solution, regularly scheduled queries to the database, comparing the current state with the previously detected state, could determine differences and thus detect updates. However, it is obvious that such an approach is harder to implement and performs worse than the active push solution using triggers. Thus, polling is no valid alternative.

The second kind of potential alternative is a modification of the information system's code itself. However, as we already illustrated in chapter 4, this is no promising alternative, even if software-technological tools such as aspect oriented programming (AOP) are used [GF05].

Thus, according to our research and to our best knowledge, there is no sensible alternative to the usage of database triggers when updates in information systems which store their data in relational databases have to be monitored.

11.1.4.3 Alternatives to Materialized Views

In the field of relational databases, there is no reasonable alternative to materialized views for the storage of event propagation indices. The application of database indices is limited to single tables and/or views, so more complex index structures, like the event propagation indices, have to be built individually - either by using materialized views, or, if the database system does not support them, by maintaining index tables using triggers to keep them up-to-date. However, this is only a technological workaround to emulate the behaviour of materialized views, so that we do not consider it an alternative to the usage of materialized views.

Hence, in our opinion, there is no reasonable alternative to materialized views for the realization of our optimization concept.

11.2 Rating of the Event-Handling Concept

On the next level of abstraction, the event-handling concept and the presented architecture are evaluated. Therefore, the requirements we identified in chapter 2 are recalled and checked for fulfilment, as well as the functional adequacy of our solution. In addition, special attention is paid to the non-functional requirements non-invasiveness, subsequent applicability and genericity, which play a central role in our work.

11.2.1 Fulfilment of Basic Postulated Requirements

In chapter 2 we postulated a total of 18 functional, technical and software-technological requirements which have to be fulfilled by an event-handling framework. In the following, we will show that our solution satisfies all of these demands.

11.2.1.1 Functional Adequacy

An important quality of a realized software solution is its functional adequacy: does the software or framework provide all desired functionalities, making it an appropriate tool for the use cases it should be used in?

This question can be answered in two steps: first, an indication that the functional adequacy is given is the discovery that all use cases that we tried to solve with our approach could be handled. One of the most detailed use cases we analyzed is Stud.IP: the previous chapters should give enough evidence that our approach is suitable for this particular use case. To further substantiate this claim, we briefly list a few use cases and give a simplified outline solution on the following pages.

One simple hypothetical use case stems from the universitary world, as Stud.IP does: using a room planning system, every lecture or event that has to be held must be assigned a suitable room. Once a room has been assigned to the lecture and the number of estimated attendants changes, the responsible person for the respective room has to be informed, so that the room schedule can possibly be modified. Figure 11.18 gives an outline how this use case could be modeled.

Figure 11.18: Solution outline for use case *Room Planning*

Another use case arises in the domain of location based services: assuming that the current location of an arbitrary user is stored in a database, he might want to be informed about all interesting "news" about the location he is currently at. In the solution outline presented in figure 11.19, we assume that there might be new or updated events, as well as special offers in shops at the various locations.

The last sample use case we examined originates in the research project *MonArch* [FS09], a digital library for the management of restauration information about ancient buildings. These buildings are hierarchically organized in a comprehensive partonomy. A possible use case in this area of application might be that any worker who restaurated an element of this partonomy (e.g. a particular brick) might want to be informed about all added or modified documents that concern this particular element of the partonomy, a sub-part of this element, or a more general element of the partonomy containing the

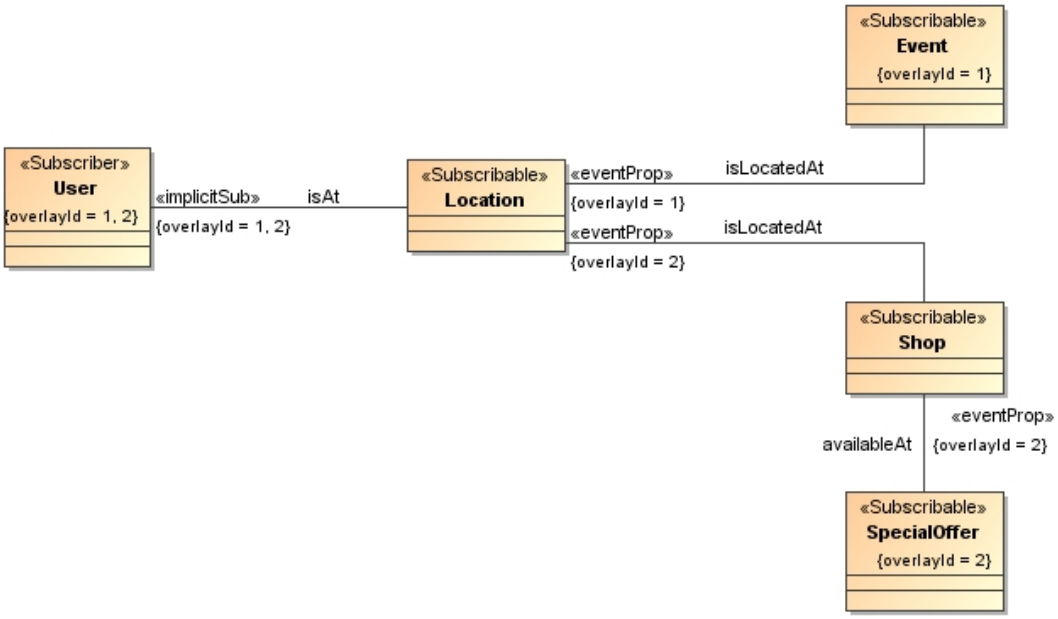


Figure 11.19: Solution outline for use case *Location Based Services*

part she or he worked on. Such a situation could for instance be realized using the design shown in figure 11.20.

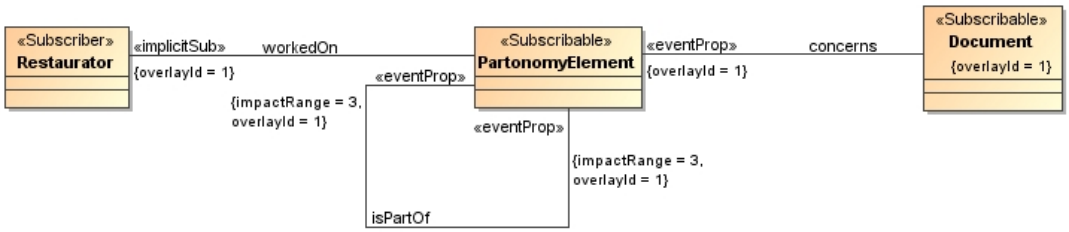


Figure 11.20: Solution outline for use case *MonArch*

Although many imaginable scenarios can be handled with our approach, it is easy to find examples that require a more complex subscription specification. For instance in the application domain of stock trading, users should maybe only be informed if the updated value (price of particular shares) differs significantly from its value before the update. Such complex conditions can - at the moment - not be handled using our approach, so that this possible improvement shold be realized in a follow-up project (see open ends in chapter 12.

However, there are more indications that our solution is suitable in many cases: in chapter 2, we derived a series of universally valid functional requirements from the use cases we analyzed. In the following, we recapitulate those requirements and check them for their fulfilment.[2]

/R1.1/ Entities must be markable as subscribable By providing the stereotype ≪Subscribable≫ in our profile, which can be applied to any persistent entity, this requirement is clearly fulfilled.

/R1.2/ Entities must be markable as subscriber Similarly, the stereotype ≪Subscriber≫ satisfies this requirement.

/R1.3/ Monitoring of subscribables must be limitable to individual attributes Obviously, the stereotype ≪ObservedAttribute≫, applicable to persistent attributes of persistent entities, has been designed to comply with this requirement.

/R1.4/ Implicit subscriptions must be supported Implicit subscriptions can be realized by applying the stereotype ≪implicitSub≫ to associations in the data model.

/R1.5/ Explicit subscriptions must be supported By providing a generic storage schema for explicit subscriptions (cf. figure 9.1) and incorporating the logic to determine all explicit subscribers of an update into the trigger code, this requirement can be considered as fulfilled, too.

/R1.6/ Transitive propagation of update events must be supported The transitive propagation of update events can be achieved by tagging the respective association ends with the stereotype ≪eventProp≫. Thus, this functional requirement is satisfied.

/R1.7/ Impact of transitive propagation must be limitable Furthermore, the stereotype ≪eventProp≫ can be parametrized with the tagged value impactRange, which is considered when determining the maximum distance between an updated object and

[2] All of the following requirements are fulfiled both by our concept as well as by our protoypic implementation. However, we will simply refer to the corresponding stereotype or tagged value, knowing that all of these stereotypes are representations of formal concepts in our approach.

all related objects which should be considered as updated, too. Accordingly, this requirement is also fulfilled.

/R1.8/ Transitive propagation along associations must be directed Since the stereotype ≪eventProp≫ has to be applied to association ends, the direction of the transitive event propagation is implicitly directed.

11.2.1.2 Technical / Architectural Adequacy

Besides the functional adequacy of our approach, its technical and architectural suitability has to be checked, too. Therefore, we take a retrospective look at the technical requirements that were identified during the use case analysis and check their fulfilment.

/R2.1/ Updates must be monitored and handled centrally in the data storage According to our system architecture (cf. fig. 4.5), the event detection layer is located centrally on top of the data storage. By creating database triggers, the prototypic implementation corresponds to the architecture blueprint, thus fulfilling this requirement.

/R2.2/ Updates must actively be detected and pushed to subscribers In our reference implementation, updates are actively detected by the generated database triggers. However, the trigger body reacts to those detections by simply storing notification tuples, but does not really push the notifications to subscribers. However, this is not a fundamental flaw of our approach, but could easily be corrected by changing the triggers' behaviour. All major database systems allow the execution of stored procedures or similar coding from the trigger body, so active publication (for instance by sending eMail) can be provided by our framework, although not implemented within the prototype.

/R2.3/ All event-handling constructs must be based on the system's data model This requirement is fulfilled obviously, as all event-handling stereotypes are applied to classes and associations representing the data model.

/R2.4/ Hierarchical structures must be supported efficiently As we showed in various examples throughout this dissertation, hierarchical event-handling structures are supported using the concept of event propagation. In combination with the usage of event propagation indices (i.e. materialized views in our reference implementation), these structures can also be handled efficiently, as our experiments (cf. section 11.1.2) showed.

/R2.5/ Precomputation of event propagation has to be used, where appropriate By introducing event propagation indices and realizing them using materialized views, this requirement has been taken into account and is thus fulfilled.

/R2.6/ The system has to adapt to different usage characteristics during lifetime Although our prototypic reference implementation does not contain any functionality to monitor the read- and write-access statistics at runtime, the proposed architecture allows for the continuous adaption of the monitoring triggers to the changed behaviour. For instance, the materialized view and trigger definitions could be modified at any time in order to respond to new access characteristics. Thus, we consider this requirement fulfilled - however, the reference implementation would have to be extended, which remains an open end of our work.

11.2.1.3 Software-Technological Adequacy

What remains to show is that the requirements that were postulated from a software-technological view are satisfied, too.

/R3.1/ Specification of event-handling semantics has to be declarative The first software-technological requirement is fulfilled implicitly: MDA is declarative by its nature. Event-handling systems can be built using our approach without writing a single line of imperative code.

/R3.2/ The system must be open to future modifications Both our architecture as well as the model-driven paradigm itself have been designed to support the often cited anticipation of change. Since this requirement is fulfilled due to the fact that we used a generative approach, this discussion will be led in more depth in section 11.3.

/R3.3/ The semantics must be open to future modifications Similarly, this requirement is satisfied because a generative solution has been implemented. Thus, this discussion is also lead in section 11.3.

/R3.4/ Concept must be applicable to existing systems in a non-invasive way As a final requirement, the claimed non-invasiveness of our approach has been fulfilled due to the combination of the generative approach with an appropriate technical architecture. This fulfilment is mainly due to the following key-point of our proposed architecture: as figure 4.5 shows, the publish-subscribe component is completely separated from the legacy application - by using database triggers that are independent of the actual information system implementation. This separation is also achieved in our prototypic implementation. Thus, the legacy system does not have to be modified to introduce event-handling functionality. However, a complete separation of the publish-subscribe component from the legacy application is not always desired: for instance, the GUI of the publish-subscribe component should possibly be integrated into the application's GUI, so that the user does not have to switch between different applications. However, this challenge is out of our focus and remains an open end of our work.

Hand in hand with the non-invasiveness goes the subsequent applicability of our event-handling concept. Besides the non-invasiveness, a second factor is important: any information that is necessary for the generation of the new components can subsequently be derived from an existing application and/or specified by developers. This can be done without the need to know about implementation details or even source code, as the process in figure 4.6 already showed.

In practice, the subsequent applicability has been proven by the use case *Stud.IP* which was successfully and subsequently enhanced with event-handling functionality. Thus, there are apparently no barriers for the retrofitting of our event-handling component to an arbitrary existing system.

11.2.2 Technological and Conceptual Alternatives

To evaluate alternatives to our event-handling concept and the proposed reference architecture, the search space in which potential alternatives could be found has to be limited. In order to be comparable to our solution, an alternative approach at least should somehow tie event-handling declarations to the data model of the information system and should thus allow a declarative specification of the desired event-handling

functionality. However, as we already showed in chapter 3, there is - to our knowledge and after intensive research - no comparable approach, so that an evaluation of conceptual alternatives is not possible.

11.3 Review of the Generative Approach in General

In retrospective, the decision to choose a generative approach can be evaluated with respect to many different criteria, like necessary development efforts, support for the anticipation of change or technical and conceptual extensibility, to name but a few. In the following, we try to give an overview of the assets and drawbacks of the generative approach we chose.

11.3.1 Assets and Drawbacks

According to our experience, the generative approach offers a lot of advantages to traditional software development:

First of all, development costs can be significantly reduced, once the domain specific language and the respective transformations have been developed and tested thoroughly. During our work, we found out that the design of an event-handling model like the one from figure 10.5 takes about 30 minutes for an experienced modeler. In contrast, the manual development of the triggers and materialized views, as presented in listings 10.3 and 10.5 takes at least two to three hours and is rather error prone. Although these results do not stem from sound statistic experiments, they give evidence that the usage of a generative approach leads to reduced development efforts in our scenario.

Furthermore, the quality of the developed software, i.e. in our case of the views and triggers, is much higher than in manually developed solutions. Since the generation and optimization process is completely automated, there is no possible error source apart from the modelling itself. Especially when the models (and thus the necessary triggers) become large and complicated, as for instance in our sample scenario from section 10.6, the ensured correctness is of great value.

However, there is one significant deficit: model driven software development can only be as flexible as the domain specific language and the transformations. New requirements, which would usually be fulfilled by a manual implementation of the desired functionality, can not be realized easily. Instead, they first have to be generalized,

parametrized and put into the meta-model and the transformations. Although this can be considered as a deficit, it is also a chance - if newly identified requirements find their way into the model driven software development framework, they are automatically available for all users of the framework. As an example, we briefly developed a strategy of how to extend our concept so that only updates of observed attributes, where the delta between new and old value extends a given threshold, trigger the event handling. In principle, all that was necessary to achieve this is the introduction of a new tagged value for observed attributes named delta and the transformation of the new condition abs (old.value - new.value) $<$ delta into the generated triggers by extending the transformation templates. Further, the disadvantage of missing flexibility can be compensated by allowing developers to modify the generated sources. If this is done adequately, e.g. by using the generation gap pattern, the whole development process remains almost as flexible as the traditional, manual software development process.

Especially if manual modifications of the generated code are not necessary, the final advantage of model-driven software development scores: if the model from which the code is generated changes, the resulting modifications of the generated software (in our case the database triggers) can be performed without any manual effort - changing the model and re-generating the artifacts is enough. Especially during the initial development of software or during rapid protoyping, this saves a great amount of work for the developers. But also during software maintenance, if information system models are modified or extended, all modifications can ideally be made by updating the specification and re-generating the artifacts only.

Since we evaluated the model-driven approach only within the context of our business case "event handling", we can not generalize all the advantages we presented above. Instead, we tried to extract a few underlying conditions, under which its usage should be encouraged.

11.3.2 Factors for the Successful Usage of the Generative Paradigm

As we learned during our work, there are a handful of factors which indicate that the generative paradigm can successfully be used. First of all, it is necessary that a domain specific language can be built at all, i.e. that one can abstract from the desired functionality, identify fixed parts and variable, parametrized parts, and derive the respective fragments of the code that has to be generated. This is usually done building a reference implementation and analyzing it thoroughly.

If an abstraction can be identified, it is important that the derived domain specific language does not contain any elements that are specific to the analyzed use case. In our scenario *Stud.IP*, for instance, stereotypes with a meaning like "attends and thus needs to be informed" are inappropriate - instead, the more universal construct of implicit subscriptions was introduced. With that in mind, the domain specific language that is developed will be usable for a variety of use cases, but of course limited to the application domain it was developed for (in our case the domain of "event-handling"). A good example for such a generic meta-model, for instance, is *MML* [HH04], a multidimensional meta model for the design of data warehouse schemata.

To reduce the necessary specification effort when using the developed framework, it is further important to re-use as many elements of existing domain specific languages (and the respective transformations) as possible. In our scenario, we saved a lot of development effort by reusing the persistence elements of AndroMDA. In addition to the reduced effort when developing the framework, the effort that users of the framework have to make is reduced, too, as they do not have to specify the same or similar information (table and column names, in our example) twice.

If possible, the benefit of the developed framework can further be increased if the concept is flexible and comprehensive enough so that the generated artifacts do not have to be modified by later users of the framework. This enables them to (re-)generate the code at any time without the fear of losing any manual modifications, and thus both encourages them to use the generators as often as possible - an important factor for the success of the framework.

The final key to success of course is not a technical, but economic one: the business case for which the domain specific language and transformation templates are developed should be as universal and as broadly applicable as possible - by using the framework as often as possible, the initial effort for its development can soon be amortized and lead to both technical and economical advantages during software development.

11.4 Summary

In this chapter, we evaluated our solution on three different levels of abstraction with respect to objective criteria. As we could show, all major requirements are fulfiled, making our proposal an appropriate solution for the given task formulation. We further discussed the pros and cons of the generative approach in general and showed under which circumstances the model-driven approach is suitable. However, many open ends

and possible starting points for subsequent research projects have been identified in the course of our work. Therefore, we conclude this dissertation with an overview of our contribution and the open ends, which is subject to the remaining last chapter.

"The best way to predict the future is to invent it."

Alan Kay

12 Contribution and Open Ends

In the previous eleven chapters, our work has been presented and many of our results have been shown. On the following pages, we will finally summarize the experiences we made, present the contribution of our work in condensed form and list some of the open ends that remain and that could be picked up by follow-up research projects.

12.1 Contribution of Our Work

In this dissertation, we developed a non-invasive approach to integrate event-handling functionality into information systems. In the course of our work, we made the following contributions.

First, we did a detailed analysis of event-handling requirements in state-of-the-art information systems, leading to a list of functional and non-functional requirements.

To fulfil these requirements, an innovative event-handling concept and a suitable architecture with the following qualities has been developed:

- The concept provides simple, yet generic constructs that can be assembled to define common event-handling semantics, together with a formal specification of their semantics.
- These constructs are based on the traditional object-oriented model.
- The proposed approach uses the model-driven design paradigm, generating event-handling code from respective models.
- To realize event-handling models, a domain specific language expressing the event-handling semantics has been developed.
- The concept and the proposed architecture contain integrated optimization facilities.
- The whole approach is platform-independent.

As a proof of concept, a reference-implementation, using MDA as development paradigm and relational databases with triggers as a target platform, has been developed and evaluated. The evaluation of our work revealed that our concept

- provides sufficient performance for productive usage,
- is functionally adequate,
- can easily be extended to suit additional needs,
- can be applied non-invasively and subsequently to legacy systems
- and fulfills most of the commonly accepted criteria of good software quality.

From a more abstract point of view, the usage of the code-generating approach in our business case was evaluated and the following advantages were identified:

- Development efforts can significantly be reduced by using model-driven software development.
- The quality of the software that is built using code generation is higher than manually developed software.
- Although software development based on domain specific models is less flexible than traditional software development, well-designed meta-models, domain specific languages and transformations give enough flexibility for the integration of new functionalities.

- The simple generation of code from a given model supports developers in quickly developing prototypes, even if the requirements for those prototypes, and thus the respective models, are subject to frequent changes during this development stage.

Finally, as a meta-result of this analysis, we worked out the following factors which should encourage the usage of a code-generating, model-driven approach, so that the above-mentioned advantages can actually be fully exploited:

- Elements of the domain specific language should be independent of any actual business case,
- existing (and maybe de-facto standard) modelling concepts should be reused, if possible and appropriate,
- ideally, code artifacts should be generated as comprehensive as possible, so that manual modifications or extensions to the generates are unnecessary
- and finally, the development framework should be as universally applicable as possible, so that the development costs for the framework amortize quickly.

Although we presented a comprehensive approach "from concept over realization to evaluation", we encountered several problems and questions. Those aspects were taken out of the focus of our work and will be listed as "open ends" in the following.

12.2 Open Ends

As final part of this dissertation, we briefly list the open ends of our work as a starting point for follow-up research projects, divided into technological improvements, conceptual improvements and visionary ideas.

12.2.1 Technological Improvements

The prototypic implementation of our concept could still be improved in various ways. A few of those potential improvements are listed in the following.

Scenario Monitoring Using Triggers As proposed by our architectural concept, the actual usage scenario of the information system's underlying database could constantly

be monitored in order to give up-to-date information to the optimizer. To achieve this, appropriate triggers detecting and counting the read and write accesses could be developed.

Auto-Tuning of the Event-Handling System If this up-to-date scenario information is present at any time, the system could further be improved by automatically adapting to a new scenario: triggers and materialized views could be redefined and recreated whenever the information system usage significantly changes. In the course of this work, existing technologies for the automatic selection and maintenance of materialized views and indices, as for instance developed by researchers at *Microsoft* [ACN01, BC06], should be evaluated and possibly used for the system's automatic tuning.

Generation of the Graphical User Interface In our reference implementation, subscriptions and notifications are simply stored in database tables. Since all required information about the subscribables and subscribers are contained in the information system's model, however, it would be possible to automatically generate a graphical user interface which serves as the users' interface for the specification of subscriptions and presentation of notifications and updated data. Further, our proposal does not integrate the event-handling GUI and functionality with the legacy application, but only with its data storage, so for a successful integration of our solution into real information systems, it has to be evaluated how the required user interface for the specification of subscriptions and presentation can be tightly integrated with the legacy system's GUI.

Asynchronous Subscriber Determination An important possibility to improve the performance of the information system's database would be to asynchronously determine all relevant subscribers of an update: triggers would only detect and store the update, while the determination of subscribers would take place in a second, decoupled phase (e.g. during batch processing at night). This strategy could improve the performance of the online database system at the cost of less timely notifications and possibly outdated information during the second phase.

Sophisticated Index Maintenance Another way how to optimize performance could be a more sophisticated way of storing event propagation indices: by using more efficient algorithms to maintain the index, like e.g. labelled spanning trees as proposed by Agrawal, Borgida and Jagadish [ABJ89], the *2-hop-approach* proposed by Cohen et

al. [CHKZ02] or the compact reachability labeling invented by He et al. [HWYY05], the determination of relevant subscribers could be sped up.

OCL Constraints From a usability point of view, the correct application of the stereotypes and tagged values could be supported better. UML profiles by default offer no possibility to restrict the usage of stereotypes according to given rules. However, the Object Management Group proposed another standard to model such constraints: the Object Constraint Language (OCL) [Obj06], which is part of the UML 2.0 specification. Using this constraint language, it would be possible to specify invariants that are required for the correct use of all event-handling constructs, such as "associations may only be tagged as event propagating, if they connect two subscribables".

12.2.2 Conceptual Improvements

Besides the implementation specific tasks, several issues on a conceptual layer remain open.

Security issues As we already detected during our evaluation, security restrictions are an important part of an event-handling system, but are completely out of the focus of our work. Thus, an important follow-up project should introduce a concept of how to integrate the event-handling specification and security mechanisms to restrict access to notifications appropriately.

Determination of Distributed Updates As we briefly sketched in section 11.1.3, the applicability of our approach is basically adequate for the application in distributed environments. However, it remains to research how the detection of events across distributed, heterogeneous data storages actually could be realized.

Optimiziation of Cyclic Path Fragments In section 6.4.3, we already explained why unnecessary, but "harmless" paths that match a path description containing cycles can occur. Thus, as an open end of our work, overall performance could be improved by implementing an algorithm that purges superfluous paths.

Complex subscription conditions Although most of the subscription use cases we encountered could be solved with our concept, there are many scenarios in which

more complex subscription conditions could be necessary. Thus, it would be a reasonable follow-up research project to further refine and extend the concept and domain specific language and show how to transform the specification automatically into the corresponding code.

12.2.3 Visionary Ideas

While the latter improvement possibilities could be realized in the short term, our work also revealed "visionary ideas" which could be elaborated within larger scale projects.

Architectural Alternatives A research project could evaluate several architectural alternatives to our solution. For instance, it would be interesting to know how data warehouses could be used to realize the event-handling functionality: regular ETL processes could extract the data to the warehouse, while the actual subscriber determination could be processed by the data warehouse, which could also contain the event-propagation indices. Of course, further architectural alternatives that we are currently not aware of could be researched - not only to find an ideal solution, but to offer different alternatives to companies who already use certain architectures and systems.

Extending SQL to Integrate Notification Aspects into the DDL Finally, a very promising idea could try to bring the event-handling concept and relational databases even closer together: by directly integrating the notification semantics into the data description language (DDL), the database system itself could be aware of the event-handling specification. For instance, a foreign key definition could be marked by an additional keyword IMPLICIT or EVENTPROP. On the one side, this would simplify the usage of our event-handling concept, on the other side, the direct consideration of the event-handling specification within the database system might lead to significant performance benefits. Thus, we would firmly recommend to further investigate this possibility.

12.3 Summary

In summary, it can be stated that our work provides a solid solution for the model-driven development of event-handling functionalities. Although many aspects have not yet been explored and many problems and questions remain open, we can certainly claim that our work provides a sound basis for the further refinement of a non-invasive, code generating framework for the development of event-handling functionalities in information systems.

A

Bibliography

[ABC+00] Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors. *Proceedings of the 26th International Conference on Very Large Data Bases*. Morgan Kaufmann, September 2000.

[ABEYH00] Asaf Adi, David Botzer, Opher Etzion, and Tali Yatzkar-Haham. Push technology personalization through event correlation. In Abbadi et al. [ABC+00], pages 643–645.

[ABJ89] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 253–262, New York, NY, USA, 1989. ACM.

[ACN01] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Materialized view and index selection tool for microsoft sql server 2000. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, page 608, New York, NY, USA, 2001. ACM.

[Anda] AndroMDA. AndroMDA hibernate cartridge. `http://galaxy.andromda.org/docs/andromda-hibernate-cartridge/index.html`. last visited: 2010-05-05.

[Andb] AndroMDA. Homepage of the AndroMDA project. `http://www.andromda.org`. last visited: 2010-05-05.

[Apa] Apache Software Foundation. Velocity. `http://velocity.apache.org/`. last visited: 2010-05-05.

[ASS+99] Marcos Kawazoe Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar Deepak Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[BBC+98] Philip A. Bernstein, Michael L. Brodie, Stefano Ceri, David J. DeWitt, Michael J. Franklin, Hector Garcia-Molina, Jim Gray, Gerald Held, Joseph M. Hellerstein, H. V. Jagadish, Michael Lesk, David Maier, Jeffrey F. Naughton, Hamid Pirahesh, Michael Stonebraker, and Jeffrey D. Ullman. The asilomar report on database research. *SIGMOD Record*, 27(4), December 1998.

[BC06] Nicolas Bruno and Surajit Chaudhuri. To tune or not to tune?: a lightweight physical design alerter. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 499–510. VLDB Endowment, 2006.

[Ber02] Phil Bernstein, editor. *Proceedings of the 28th International Conference on Very Large Data Bases*. Morgan Kaufmann, August 2002.

[BF06] Hanen Belhaj-Frej. Personnalisation services for digital libraries. Presented at the ECDL 2006 - Doctoral Consortium, September 2006.

[BFRS06] H. Belhaj-Frej, P. Rigaux, and N. Spyratos. User notification in taxonomy based digital libraries. *Proceedings of the 24th annual ACM international conference on Design of communication*, pages 180–187, 2006.

[BKK04] Martin Bernauer, Gerti Kappel, and Gerhard Kramler. Composite events for xml. In *Proceedings of WWW2004*, New York, USA, May 17–22 2004. ACM.

[BKLW99] Susanne Busse, Ralf-Detlef Kutsche, Ulf Leser, and Herbert Weber. Federated information systems: Concepts, terminology and architectures. Technical Report Forschungsberichte des Fachbereichs Informatik 99-9, TU Berlin, 1999.

[BLS03] Don Batory, Jia Liu, and Jacob Neal Sarvela. Refinements and multidimensional separation of concerns. *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 48–57, 2003.

[CAW98] Sudarshan S. Chawathe, Serge Abiteboul, and Jennifer Widom. Representing and querying changes in semistructured data. In *Proceedings of the 14th International Conference on Data Engineering (ICDE'98)*, page 4, 1998.

[CBB03] M. Cilia, C. Bornhövd, and A. P. Buchmann. Cream: An infrastructure for distributed heterogeneous event-based applications. In *Proceedings of the International Conference on Cooperative Information Systems*, pages 482–502. Springer, 2003.

[CCW00] Stefano Ceri, Roberta Cochrane, and Jennifer Widom. Practical applications of triggers and constraints: Success and lingering issues (10-year award). In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 254–262. Morgan Kaufmann, 2000.

[CG07] K. Mani Chandy and Dieter Gawlick. Event processing using database technology. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1169–1170, New York, NY, USA, 2007. ACM.

[CGL+97] Sudarshan Chawathe, Vineet Gossain, Xiang Liu, Jennifer Widom, and Serge Abiteboul. Change management in heterogeneous semistructured databases (demonstration description). Technical report, Computer Science Department, Stanford University, 1997.

[CHKZ02] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *SODA '02: Proceedings of*

the thirteenth annual ACM-SIAM symposium on Discrete algorithms, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[CM94] Sharma Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[CMD02] Keith Cheverst, Keith Mitchell, and Nigel Davies. Exploring context-aware information push. *Personal and Ubiquitous Computing*, 6(4):276–281, 2002.

[CN02] Junghoo Cho and Alexandros Ntoulas. Effective change detection using sampling. In Bernstein [Ber02], pages 514–525.

[Con02] Stefan Conrad. Integration heterogener Datenbestände. Heinrich-Heine-Universität Düsseldorf. *Jahrbuch*, pages 189–201, 2002.

[Cor] IBM Corporation. DB2 Online Documentation. `http://publib.boulder.ibm.com/infocenter/db2help/index.jsp`. last visited: 2010-05-05.

[CRW98] A. Carzaniga, D. Rosenblum, and A. Wolf. Design of a scalable event notification service: Interface and architecture. Tech. Rep. CU-CS-863-98, Dept. of Computer Science, Univ. of Colorado at Boulder, September 1998.

[Dat] Data-Quest GmbH. Homepage of the Stud.IP project. `http://www.studip.de`. last visited: 2010-05-05.

[DB92] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 107–114, New York, NY, USA, 1992. ACM Press.

[Deu04] Deutsches Patent- und Markenamt. In einer relationalen Datenbank integriertes Contextbasiertes System zur Veröffentlichung und Abonnierung. German Translation of the European Patent G06F 17/30, 06 2004.

[DL05] Eric Dombrowski and Jens Lechtenbörger. Evaluation objektorientierter Ansätze zur Data-Warehouse-Modellierung. *Datenbank-Spektrum*, 15, 2005.

[Ecl] Eclipse Foundation. Homepage of AspectJ. `http://www.eclipse.org/aspectj/`. last visited: 2010-05-05.

[EDS+04] G. Edwards, G. Deng, D. Schmidt, A. Gokhale, and B. Natarajan. Model-driven configuration and deployment of component middleware publisher /subscriber services. In *Proc. of the 3rd ACM Int. Conference on Generative Programming and Component Engineering*, October 2004.

[EFGK03] The Eugster, P.A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe, 2003.

[EG01] Patrick Th. Eugster and Rachid Guerraoui. Content-Based Publish/-Subscribe with Structural Reflection. In *Proceedings of the 6th Usenix Conference on Object-Oriented Technologies and Systems*. The USENIX Association, 2001.

[EGD01] Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On objects and events. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 254–269, New York, NY, USA, 2001. ACM Press.

[EN04] Ramirez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 2004.

[FdSR03] Roberto S. Silva Filho, Cleidson R. B. de Souza, and David F. Redmiles. The design of a configurable, extensible and dynamic notification service. In *DEBS '03: Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8, New York, NY, USA, 2003. ACM Press.

[FFM01] Sergio Flesca, Filippo Furfaro, and Elio Masciari. Monitoring web information changes. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC '01)*, page 421, 2001.

[FJL+01] Françoise Fabret, H. Arno Jacobsen, François Llirbat, Joăo Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 115–126, New York, NY, USA, 2001. ACM.

[FMG02] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. A modular approach to build structured event-based systems. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 385–392, New York, NY, USA, 2002. ACM.

[Fra03] David S. Frankel. *Model Driven Architecture - Applying MDA to Enterprise Computing*. Wiley Publishing Inc., 2003.

[FRS04] Hanen Belhaj Frej, P. Rigaux, and Nicaloas Spyratos. Notification and recommendation services for web communities. Presented at the 2nd IST Workshop on Metadata Management in Grid and P2P Systems (MMGPS): Models, Services and Architectures, December 2004.

[FRS06] Hanen Belhaj Frej, Philippe Rigaux, and Nicolas Spyratos. Matching algorithms for user notification in digital libraries. In Dominique Laurent, editor, *BDA*, 2006.

[FS09] Burkhard Freitag and Christoph Schlieder. Monarch - digital archives for monumental buildings. *Künstliche Intelligenz*, 4:30–35, 2009.

[FZB+04] Ludger Fiege, Andreas Zeidler, Alejandro Buchmann, Roger Kilian-Kehr, and Gero Mühl. Security aspects in publish/subscribe systems, 2004.

[GBD05] Heinz Lothar Grob, Frank Bensberg, and Blasius Lofi Dewanto. Model driven architecture (mda): Integration and model reuse for open source platforms. `http://eleed.campussource.de/archive/1/81/` (Downloaded: 2006/01/25), 2005.

[Gen] Gentleware AG. Poseidon for UML. `http://www.gentleware.com`. last visited: 2010-05-05.

[GF05] Michael Guppenberger and Burkhard Freitag. Intelligent Creation of Notification Events in Information Systems - Concept, Implementation and Evaluation. In Abdur Chowdhury and et. al., editors, *Proc. of the 14th ACM Int. Conference on Information and Knowledge Management*, pages 52–59. ACM, ACM Press, November 2005.

[GKP99] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the ready event notification service. In *Proc. of the 19th IEEE Int. Conference on Distributed Computing Systems Middleware Workshop*, page 108, 1999.

[GM99a] Ashih Gupta and Inderpal Singh Mumick, editors. *Materialized Views - Techniques, Implementation, and Applications.* MIT Press, 1999.

[GM99b] Ashish Gupta and Inderpal Singh Mumick. Introduction to views. In *Materialized Views - Techniques, Implementations, and Applications* [GM99a], pages 3–8.

[Gup01] Michael Guppenberger. Konzeption und Implementierung eines taxonomie-basierten Wissensmanagementsystems. Master's thesis, University of Passau, 2001.

[HDG+07] Mingsheng Hong, Alan J. Demers, Johannes E. Gehrke, Christoph Koch, Mirek Riedewald, and Walker M. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 761–772, New York, NY, USA, 2007. ACM.

[HH04] Olaf Herden and Arne Harren. Die ODAWA-Methodik für den Entwurf von Data-Warehouse-Datenbanken. *Informatik - Forschung und Entwicklung*, 19(2):87–96, 2004.

[HMJ+96] Robert W. Hall, Amit Mathur, Farnam Jahanian, Atul Prakash, and Craig Rassmussen. Corona: a communication service for scalable, reliable group collaboration systems. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 140–149, New York, NY, USA, 1996. ACM Press.

[HR01] Theo Härder and Erhard Rahm. *Datenbanksysteme - Konzepte und Techniken der Implementierung.* Springer, 2001.

[HSF+05] Otthein Herzog, Hans-Jörg Schek, Norbert Fuhr, Abdur Chowdhury, and Wilfried Teiken, editors. *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005.* ACM, 2005.

[HV02a] A. Hinze and A. Voisard. Composite events in notification services with application to logistics support. Technical Report tr-B-02-10, Freie Universitaet Berlin, 2002.

[HV02b] Annika Hinze and Agnes Voisard. A parameterized algebra for event notification services. In *Proceedings of the 9th International Symposium on Temporal Representation and Reasoning (TIME 2002), Manchester,*

UK, 2002.

[HWYY05] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Compact reachability labeling for graph-structured data. In Herzog et al. [HSF+05], pages 594–601.

[IBM] IBM. Homepage of IBM's DB2. `http://www-306.ibm.com/software/data/db2/`. last visited: 2010-05-05.

[IK05] Yannis E. Ioannidis and Georgia Koutrika. Personalized systems: Models and methods from an ir and db perspective. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB*, page 1365. ACM, 2005.

[Int] Intland Software. codeBeamer - Collaborative Software Development Solution. `http://www.intland.com/products/codebeamer.html`. last visited: 2010-05-05.

[KdM05] Martin Kempa and Zoltán Ádám Mann. Model driven architecture. *Informatik Spektrum*, 28(4):298–302, 2005.

[KF07] Stephan Kiemle and Burkhard Freitag. Providing context-sensitive access to the earth observation product library. *Research and Advanced Technology for Digital Libraries (ECDL 2007)*, 4675:223–234, 2007.

[KGF06] Stefan Kurz, Michael Guppenberger, and Burkhard Freitag. A UML profile for modeling schema mappings. In John F. Roddick, V. Richard Benjamins, Samira Si-Said Cherfi, Roger Chiang, Ramez Elmasri, Hyoil Han, Martin Hepp, Miltadis Lystras, Vojislav B. Misic, Geert Poels, Il-Yeol Song, Juan Trujillo, and Christelle Vangenot, editors, *Advances in Conceptual Modeling - Theory and Practice*, volume 4231 of *Lecture Notes in Computer Science*, pages 53–62. Springer, November 2006.

[Kie02] Stephan Kiemle. From digital archive to digital library - a middleware for earth-observation data management. *Research and Advanced Technology for Digital Libraries, 6th European Conference, ECDL 2002 Proceedings*, 2458:230–237, 2002.

[KPdLB03] Manuele Kirsch-Pinheiro, José Valdeni de Lima, and Marcos R. S. Borges. A framework for awareness support in groupware systems. *Comput. Ind.*, 52(1):47–57, 2003.

[KR95] Balachander Krishnamurthy and David S. Rosenblum. Yeast: A general purpose event-action system. *IEEE Trans. Software Eng.*, 21(10):845–857, 1995.

[KWB04] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise.* Addison Wesley, 2004.

[LMTS02] Sergio Luján-Mora, Juan Trujillo, and Il-Yeol Song. Multidimensional modeling with uml package diagrams. In *ER '02: Proceedings of the 21st International Conference on Conceptual Modeling*, pages 199–213, London, UK, 2002. Springer-Verlag.

[LPT+98] Ling Liu, Calton Pu, Wei Tang, David Buttler, John Biggs, Tong Zhou, Paul Benninghoff, Wei Han, and Fenghua Yu. CQ: a personalized update monitoring toolkit. In *Proceedings of the ACM Sigmod Conference*, pages 547–549, 1998.

[LSL00] Minsoo Lee, Stanley Su, and Herman Lam. Event and rule services for achieving a web-based knowledge network. Technical report, University of Florida, Dpt. of Computer and Information Science and Engineering, 2000.

[LSL04] Minsoo Lee, Stanley Su, and Herman Lam. Event and rule services for achieving a web-based knowledge network. *Knowledge-Based Systems*, 17:179–188, 2004.

[LVA+99] Wen-Syan Li, Quoc Vu, Divakant Agrawa, Yoshinori Hara, and Hajime Takano. Powerbookmarks: a system for personalizable web information organization, sharing, and management. In *Computer Networks*, volume 31, pages 1375–1389. Elsevier Science, 1999.

[MCB03] D. Scott McCrickard, Mary Czerwinski, and Lyn Bartram, editors. *International Journal of Human-Computer Studies*, volume 58. Elsevier, May 2003.

[MCSN03] D. Scott McCrickard, C. M. Chewar, Jacob P. Somervell, and Ali Ndiwalana. A model for notification systems evaluation - assessing user goals for multitasking activity. *ACM Trans. Comput.-Hum. Interact.*, 10(4):312–338, 2003.

[MFP06] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[MG02] Jonathan P. Munson and Vineet K. Gupta. Location-based notification as a general-purpose service. In Marisa S. Viveros, Hui Lei, and Ouri Wolfson, editors, *Workshop Mobile Commerce*, pages 40–44. ACM, 2002.

[Müh02] Gero Mühl. Large-scale content-based publish/subscribe systems, 2002.

[Mica] Microsoft Corporation. Homepage of Visio. `http://office.microsoft.com/visio`. last visited: 2010-05-05.

[Micb] Microsoft Corporation. Microsoft TechNet: SQL Server 2005 Notification Services. `http://technet.microsoft.com/en-us/sqlserver/bb331774.aspx`. last visited: 2010-05-05.

[Micc] Microsoft Corporation. SQL Server 2005. `http://www.microsoft.com/sql/prodinfo/default.mspx`. last visited: 2010-05-05.

[MSUW04] Stephen Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled*. Addison Wesley, 2004.

[MTSP05] Jose-Norberto Mazon, Juan Trujillo, Manuel Serrano, and Mario Piattini. Applying mda to the development of data warehouses. In *DOLAP '05: Proceedings of the 8th ACM international workshop on Data warehousing and OLAP*, pages 57–66, New York, NY, USA, 2005. ACM.

[MZV07] Tova Milo, Tal Zur, and Elad Verbin. Boosting topic-based publish-subscribe systems with dynamic clustering. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 749–760. ACM, 2007.

[NoM] NoMagic Inc. MagicDraw. `http://www.magicdraw.com/`. last visited: 2010-05-05.

[Obja] Object Management Group. Meta object facility (mof) 2.0 core specification. `http://www.omg.org/cgi-bin/doc?ptc/04-10-15.pdf`.

[Objb] Object Management Group. OMG Model Driven Architecture Homepage. `http://www.omg.org/mda/`. last visited: 2010-05-05.

[Objc] Object Management Group. The Object Management Group Homepage. http://www.omg.org. last visited: 2010-05-05.

[Objd] Object Management Group. Uml 1.4.2 specification. http://www.omg.org/cgi-bin/doc?formal/04-07-02.

[Obje] Object Management Group. The unified modeling language resource page. http://www.uml.org. last visited: 2010-05-05.

[Objf] Object Management Group. XML Metadata Interchange 2.1.1. http://www.omg.org/cgi-bin/doc?formal/2007-12-01.

[Obj04a] Object Management Group. Uml 2.0 infrastructure specification. www.omg.org/docs/ptc/03-09-15.pdf, 2004.

[Obj04b] Object Management Group. Uml 2.0 superstructure specification. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02, 2004.

[Obj06] Object Management Group. Object Constraint Language - OMG Available Specification. http://www.omg.org/spec/OCL/2.0/PDF, 2006.

[O'R05] Tim O'Reilly. What Is Web 2.0? http://www.oreilly.de/artikel/web20.html as of 2008-06-29, September 2005.

[PFJ+01] João Pereira, Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, and Dennis Shasha. Webfilter: A high-throughput xml-based publish and subscribe system. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB*, pages 723–724. Morgan Kaufmann, September 2001.

[PFL+00] João Pereira, Françoise Fabret, François Llirbat, Radu Preotiuc-Pietro, Kenneth A. Ross, and Dennis Shasha. Publish/subscribe on the web at extreme speed. In Abbadi et al. [ABC+00], pages 627–630.

[PFLS00] Joao Pereira, Francoise Fabret, Francois Llirbat, and Dennis Shasha. Efficient matching for web-based publish/subscribe systems. In *Conference on Cooperative Information Systems*, pages 162–173, 2000.

[PH03] Jörg Pleumann and Stefan Haustein. A model-driven runtime environment for web applications. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 190–204. Springer,

2003.

[PM07] Oscar Pastor and Juan Carlos Molina. *Model-Driven Architecture in Practice*. Springer Verlag, 2007.

[PS02] Peter R Pietzuch and Brian Shand. A framework for object-based event composition in distributed systems, June 2002. Presented at the 12th PhDOOS Workshop (ECOOP'02).

[QKC05] Christoph Quix, David Kensche, and Mohamed Amine Chatti. Rollenbasierte Metamodellierung zur Datenintegration. *Datenbank-Spektrum*, 5(15):5–11, 2005.

[Rau96] Uwe Rauschenbach. Supporting awareness in shared workspaces using relevance-dependent event notifications. In *Proceedings of CVE'96 Workshop Collaborative Virtual Environments*, Nottingham, September 1996.

[RCHM02] Ralf Rantzau, Carmen Constantinescu, Uwe Heinkel, and Holger Meinecke. Champagne: Data change propagation for heterogeneous information systems. In Bernstein [Ber02], pages 1099–1102.

[RJK+05] Shariq Rizvi, Shawn R. Jeffery, Sailesh Krishnamurthy, Michael J. Franklin, Nathan Burkhart, Anil Edakkunni, and Linus Liang. Events on the edge. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 885–887, New York, NY, USA, 2005. ACM.

[RKCD01] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. of the 3rd International Workshop on Networked Group Communication*, pages 30–43, London, November 2001.

[RLS05] Sebastian Richly, Wolfgang Lehner, and Daniel Schaller. GignoMDA - modellgetriebene Entwicklung von Datenbankanwendungen. *Datenbank-Spektrum*, 5(15):12–17, 2005.

[RW97] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 344–360. Springer–Verlag, 1997.

[SAB+00] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content based routing with elvin4. In *Proceedings AUUG2K*, Canberra, Australia, June 2000.

[SPD92] Stefano Spaccapietra, Christine Parent, and Yann Dupont. Model independent assertions for integration of heterogeneous sources. *VLDB Journal*, 1:81–126, 1992.

[TAJ04] David Tam, Reza Azimi, and Hans-Arno Jacobsen. Building content-based publish/subscribe, 2004.

[Tan03] Wei Tang. *Internet-Scale Information Monitoring: A Continual Query Approach*. PhD thesis, Georgia Institute of Technology, College of Computing, 2003.

[TBF+03] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe, 2003.

[Thu00] Karsten Thurow. Ein generisches Notifikations-Framework. Master's thesis, Universität Hamburg, 2000.

[TOHSMS99] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 107–119, 1999.

[TRP+04] Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr, and Jussi Myllymaki. Implementing a scalable xml publish/subscribe system using relational database systems. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 479–490, New York, NY, USA, 2004. ACM.

[Uni] University of Passau. InteLeC Homepage. `http://www.intelec.uni-passau.de/intelec_projekt.0.html`. last visited: 2008-07-06.

[VBM05] L. Vargas, J. Bacon, and K. Moody. Integrating databases with publish/subscribe. *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*, pages 392–397, June 2005.

[vdR08] Reind P. van de Riet. Twenty-five years of mokum: For 25 years of data and knowledge engineering: Correctness by design in relation to mde1the acronym mda is a trademark of the omg; mde is the al-

ternative.1 and correct protocols in cyberspace. *Data Knowl. Eng.*, 67(2):293–329, 2008.

[Wan08] Jun Wang. An interface-based modular approach for designing distributed event-based systems. Master's thesis, University of Waterloo, 2008.

[WBL+06] Shahtab Wahid, Saurabh Bhatia, Jason Chong Lee, Manuel Perez-Quinones, and D. Scott McCrickard. Cabn: Disparate information management through context-aware notifications. In *Proceedings of the 16th International Conference on Computer Theory and Applications (ICCTA)*, pages 46–49, September 2006.

[WC96] Jennifer Widom and Stefano Ceri. *Active Database Systems - Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.

[Wika] Wikimedia Foundation Inc. ISO 9126. `http://en.wikipedia.org/wiki/ISO_9126`. last visited: 2010-05-05.

[Wikb] Wikimedia Foundation Inc. Wikipedia. `http://www.wikipedia.org`. last visited: 2010-05-05.

[WJL04] Jinling Wang, Beihong Jin, and Jing Li. An ontology-based publish/-subscribe system. *Lecture Notes in Computer Science*, 3231:232–253, 2004.

[Yah] Yahoo! Inc. Flickr. `http://www.flickr.com`. last visited: 2010-05-05.

[Zei04] Andreas Zeidler. A distributed publish/subscribe notification service for pervasive environments. Dissertation, TU Darmstadt, 2004.

[ZH05] Rongmei Zhang and Y. Charlie Hu. Hyper: A hybrid approach to efficient content-based publish/subscribe. *Distributed Computing Systems, International Conference on*, 0:427–436, 2005.

[ZMU97] Detlef Zimmer, Axel Meckenstock, and Rainer Unland. A general model for event specification in active database management systems. In *Proceedings of the 5th International Conference on Deductive and Object-Oriented Databases, DOOD'97*, 1997.

[ZU96] Robert Zhang and Elizabeth Unger. Event specification and detection. Technical Report TR CS-96-8, Department of Computing and Information Sciences, Kansas State University, June 1996.

B

Technical Details

In this part of the appendix we present some technical, implementation-specific details for the sake of completeness.

B.1 Automatically Refreshing Materialized Views in DB2

In chapter 10 we explained how to use materialized views to realize the event propagation indices. Usually, the declaration of such a view would be done using REFRESH IMMEDIATE so that the view is refreshed after every modification of an underlying table and thus up-to-date whenever it is queried. However, due to restrictions in DB2 (c.f. documentation about the CREATE TABLE statement in the DB2 documentation [Cor]), immediate refreshing is not possible whenever the underlying query contains self-joins. We also tried several different approaches that did not work: using triggers in combination with stored procedures to refresh the materialized view as well as dropping and recreating the view from a trigger. None of them works due to DB2s restrictions.

To overcome these restrictions, a stored procedure has to be used in combination with an extra trigger. Listing B.1 shows the source code of this stored procedure, refreshing

a materialized view with a given name by executing the appropriate REFRESH TABLE statement.

```
CREATE PROCEDURE refresher (IN tablename VARCHAR(255))
MODIFIES SQL DATA
EXTERNAL ACTION
NOT DETERMINISTIC
LANGUAGE SQL
BEGIN
 DECLARE v_s varchar(255);
 SET v_s = CONCAT('REFRESH␣TABLE␣', tablename);
 EXECUTE IMMEDIATE v_s;
end
```

Listing B.1: Procedure for materialized view maintenance in DB2

As a counterpart, a regular update trigger like the one in listing B.2 has to be added for every table that influences the materialized view.

```
CREATE TRIGGER refreshme<tabName>
AFTER UPDATE OF <attributes>
             ON <tableName>
FOR EACH STATEMENT
BEGIN ATOMIC
  CALL refresher ('<tableName>');
end
```

Listing B.2: Trigger to call the routine for view refreshment

In addition, whenever materialized views are used, it is important in which order the triggers are created. DB2 processes triggers in the order of their creation, so the following order is necessary to get correct results:

1. Create trigger(s) to refresh views
2. Create trigger(s) accessing views for event-handling

B.2 Maintaining Event Propagation Indices with SQL Server

Although SQL Server 2005 [Micc] supports materialized views (called *indexed views*), like DB2 does, they cannot be automatically maintained if they contain self-referencing joins. To nevertheless realize the event propagation indices for the empirical validation in section 7.8.5, the index was programmed using regular database tables. Maintenance of the index was realized using refreshing triggers. Listing B.3 exemplarily shows a trigger for index maintainance.

```
CREATE TRIGGER epiA1 ON join2A
AFTER INSERT, UPDATE, DELETE AS
BEGIN
  DELETE FROM epiA;
  INSERT INTO epiA (le, ri)
     SELECT c1a.id AS le, join3A.RI AS ri
     FROM c1a AS c1a, c2a AS c2a, c2a AS c22a, join2A, join3A
     WHERE c1a.ID = join2A.LE
       AND join2A.RI = c2a.ID
       AND join3A.LE = c2a.ID
       AND join3A.RI = c22a.id;
END
```

Listing B.3: Exemplary trigger to refresh event propagation index in SQL Server

C

List of Figures

1.1 Use case: Clients accessing and modifying data 4
1.2 Overlapping of individually maintained data 4
1.3 Requirements of an event-handling framework 5

2.1 Screenshot: Lecture overview in *Stud.IP* 12
2.2 Screenshot: Notification settings in *Stud.IP* 13
2.3 Excerpt from *Stud.IP*'s data model 15
2.4 Update distribution during semester break 17
2.5 Write access during semester break 18
2.6 Update distribution during ongoing semester 19
2.7 Write access during ongoing semester 20
2.8 Screenshot: Taxonomy model in *InfoWiss* 24
2.9 Specification of data clouds in *InfoWiss* 25
2.10 Excerpt from *InfoWiss* data model 26

3.1 Generic publish/subscribe architecture by Eugster et. al. [EFGK03] . 38
3.2 Detailed architecture overview 40

4.1 Event-handling as a cross cutting concern 58
4.2 Necessity for a multitude of database triggers 59

4.3 Illustration of the declarative-generative approach 62
4.4 Dimensions of abstraction 64
4.5 System architecture 66
4.6 Designtime lifecycle 69
4.7 Designtime lifecycle in the large 70
4.8 Runtime lifecycle 71

5.1 Sample data model 74
5.2 Sample system instance 76
5.3 Sample event semantics 77
5.4 Impact of event semantics on the system instance 78
5.5 Sample graph representation 83
5.6 Sample system instance graph representation 84
5.7 Sample overlay graph representation 85
5.8 Sample graph representation to illustrate path definitions 88
5.9 Real-life data model 93
5.10 First overlay 93
5.11 Second overlay 94
5.12 Sample instance graph 95
5.13 Graph representation of data model 96
5.14 Graph representation of first overlay 96
5.15 Graph representation of second overlay 96
5.16 Sample overlay for family tree notification 101

6.1 Generic generation procedure 104
6.2 Sample overlay 111
6.3 Sample overlay leading to ping-pong updates 115

7.1 Sample path descriptions 118
7.2 Event propagation indices 119
7.3 Hierarchy of probability models 123
7.4 Hierarchy of cost models 128
7.5 Join order expressed by ϕ 130
7.6 Alternative usage of indices 133
7.7 Graphical representation of scenarios 134
7.8 Graphical representation of scenario 1 135
7.9 Graphical representation of scenario 2 136
7.10 Graphical representation of scenario 3 137
7.11 Graphical representation of scenario 4 138

7.12 Recursive construction of valid path partitionings 141
7.13 Average time per update in highly connected graph, few read and many update transactions (logarithmic scale) 143
7.14 Average time per update in highly connected graph, many read and few update transactions (logarithmic scale) 144
7.15 Average time per update in sparsely connected graph, few read and many update transactions 145
7.16 Average time per update in sparsely connected graph, many read and few update transactions 146

8.1 MDA development process 153
8.2 Sample class diagram in UML 156
8.3 Model, language and meta-model 156
8.4 Sample UML profile 157
8.5 Sample UML model using a UML profile 158

9.1 Data model for explicit subscriptions and notifications 169
9.2 Architectural view: Triggers 170
9.3 Cardinalities for associated entities 171
9.4 Relational model of 1:1 association 171
9.5 Relational model of 1:n association 172
9.6 Relational model of m:1 association 172
9.7 Relational model of m:n association 173
9.8 Subscribable entity with simple and complex observed attributes . . . 174
9.9 Normalized representation of entity with simple and complex attributes 174
9.10 Sample entity-relationship model 178

10.1 UML profile for the MDA implementation using active databases . . . 196
10.2 UML template model containing explicit subscriptions and notifications 200
10.3 Metafacades and helper classes for model-to-code transformation . . . 202
10.4 UML model of Stud.IP 214
10.5 Stereotyped UML model for Stud.IP's event-handling 215
10.6 Entity-relationship model for Stud.IP's object-oriented data model . . 225

11.1 UML model used to measure the efficiency of our approach 233
11.2 Scalability with number of users - ImpactRange 1 235
11.3 Scalability with number of users - ImpactRange 3 236
11.4 Scalability with number of users - ImpactRange 5 237
11.5 Scalability with degree of taxonomy - ImpactRange 1 238
11.6 Scalability with degree of taxonomy - ImpactRange 3 239

11.7 Scalability with degree of taxonomy - ImpactRange 5 240
11.8 Scalability with depth of taxonomy - ImpactRange 1 241
11.9 Scalability with depth of taxonomy - ImpactRange 3 242
11.10 Scalability with depth of taxonomy - ImpactRange 5 243
11.11 Benefit of optimization (Dimension: depth of taxonomy) 244
11.12 Benefit of optimization (Dimension: degree of taxonomy) 245
11.13 Benefit of optimization (Dimension: user count) 246
11.14 Performance of updates leading to notifications only 247
11.15 Performance of updates leading to notifications and view updates . . . 248
11.16 Applicability in distributed environments 250
11.17 Integration of heterogeneous data models 251
11.18 Solution outline for use case *Room Planning* 254
11.19 Solution outline for use case *Location Based Services* 255
11.20 Solution outline for use case *MonArch* 255

D

Listings

8.1 Sample transformation template 158
8.2 Sample transformation result 158
8.3 Trigger syntax in DB2 161
8.4 Sample triggers 162
8.5 Sample view definition 162
8.6 Sample view usage 163
8.7 Simplified syntax to create materialized views in DB2 163

9.1 Triggers to monitor attributes 174
9.2 Sample query to determine explicit subscribers 176
9.3 Sample query to determine implicit subscribers 177
9.4 Triggers for sample entity-relationship model 178
9.5 SQL query to read notifications for a particular user 183
9.6 Triggers for sample entity-relationship model using SQL UNION 184
9.7 Sample statement to create a materialized view as an event propagation index 188
9.8 Query to determine subscribers, using event propagation index 188
9.9 Optimized triggers for sample entity-relationship model 188

10.1 Transformation template to create materialized views 209

10.2 Transformation template to create triggers 210
10.3 Materialized view definition resulting from sample model 213
10.4 Triggers to refresh materialized views 218
10.5 Trigger definition resulting from sample model 220

B.1 Procedure for materialized view maintenance in DB2 288
B.2 Trigger to call the routine for view refreshment 288
B.3 Exemplary trigger to refresh event propagation index in SQL Server . 289