Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Managing and Modeling Persistent Data Access in Process-Driven SOAs

DISSERTATION

zur Erlangung des akademischen Grades

Doktor/in der technischen Wissenschaften

eingereicht von

Dipl.Ing. Christine Mayr
Matrikelnummer 9725295

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram DUSTDAR

Diese Dissertation haben begutachtet:

(Univ.Prof. Dr. Schahram DUSTDAR)	(Univ.Prof. Dr.rer.nat. Uwe ZDUN)

Wien, 01.02.2012

(Dipl.Ing. Christine Mayr)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Managing and Modeling Persistent Data Access in Process-Driven SOAs

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor/in der technischen Wissenschaften

by

Dipl.Ing. Christine Mayr
Registration Number 9725295

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram DUSTDAR

The dissertation has been reviewed by:

(Univ.Prof. Dr. Schahram DUSTDAR)

(Univ.Prof. Dr.rer.nat. Uwe ZDUN)

Wien, 01.02.2012

(Dipl.Ing. Christine Mayr)

Erklärung zur Verfassung der Arbeit

Dipl.Ing. Christine Mayr
Quadenstrasse 138/2, A-1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

First of all, I want to thank Prof. Dr. Schahram Dustdar and Prof. Dr. Uwe Zdun for guiding me through this thesis. I thank Prof. Dr. Schahram Dustdar for giving me the chance to do my doctorate externally under his supervision. Thank you for your professional scientific support, your ideas for improvement, and your prompt feedback whenever I had questions. I would like to say a special thanks to Prof. Dr. Uwe Zdun for his excellent scientific mentoring and advice, but also for his constructive ideas and helpful stimulations throughout this thesis. Without both of you this work would not have been possible!

I thank all the people (presently and formerly) from the distributed systems group for comments and supports, in particular, I thank Dr. Huy Tran for his fundamental works on the view-based modeling framework.

Next, I want to thank the anonymous reviewers for numerous critical comments and insights extremely helpful for building up this work.

With all my heart I thank my husband, Thomas, for his patience, encouragement, and emotional support.

Finally, I deeply thank my parents for their all-encompassing support in my whole life.

*T*o Thomas and Laurin.

Abstract

In process-driven service oriented environments, process activities in a business process can invoke services to fulfill certain tasks. A service can e.g. invoke other services, perform business logic, or read and write data from persistent data storages. In particular, in long-running business processes, which require human interaction with users, process activities often need to read or write persistent data. Commonly this persistent data access is encapsulated by a special type of services – the Data Access Services (DAS). With these DAS, business processes can access persistent data both technology-independently and database-neutrally.

Unfortunately, these DAS are not sufficiently integrated into process-oriented environments, yet. Accordingly, the relationships between the different concerns of a process such as the process activities, the DAS, the underlying object-relational mappings (ORM), and the persistent data storages are not well-defined, yet. Moreover, different stakeholders have to be able to focus on these different concerns of the process selectively. For example, in order to prevent, detect, and solve structural problems in business processes such as deadlocks, stakeholders need to have a tailored views into the persistent data access of the process. When the number of process activities grows along with the number of DAS, finding certain DAS of a process can become an impossible task for the stakeholders.

In this thesis we focus on better integrating persistent data access into process-driven service-oriented environments. In order to achieve this, firstly, we introduce the view-based data modeling framework (VbDMF) designed to specify persistent data access of processes. Based on VbDMF, we illustrate the concept of persistent data access flows used to prevent, detect, and solve structural problems in processes. In order to be able to better maintain and reuse persistent data access, we present an integral architectural approach to manage our VbDMF models and model instances: The view-based model-driven data access architecture (VMDA). Along with our concepts, we illustrate the applicability by a number of industrial case studies. Furthermore, we prove the complexity and correctness of the presented algorithms. Finally, we quantitatively show, that the VMDA returns with acceptable response times.

Kurzfassung

In prozessgetriebenen, dienstorientierten Umgebungen rufen Prozessaktivitäten von Geschäftsprozessen Dienste auf, um bestimmte Aufgaben zu erfüllen. Ein Dienst kann z.B. andere Dienste aufrufen, Geschäftslogik ausführen oder Daten von einem persistenten Speicherort lesen und schreiben. Insbesondere in langandauernden Geschäftsprozessen, die menschliche Interaktion mit Benutzern erfordern, müssen Prozessaktivitäten häufig persistente Daten lesen oder schreiben. Persistente Datenzugriffe werden heute üblicherweise von einer speziellen Art von Diensten gekapselt, den Datenzugriffsdiensten (DAS). Mittels dieser Datenzugriffsdienste können Geschäftsprozesse technologieunabhängig und datenbankneutral auf persistente Daten zugreifen.

Leider sind diese Datenzugriffsdienste noch ungenügend in prozessorientierte Umgebungen integriert. Die Beziehungen zwischen verschiedenen Belangen eines Prozesses wie den Prozessaktivitäten, den Datenzugriffsdiensten, den darunterliegenden objektrelationalen Abbildungen und den persistenten Datenspeichern sind bis dato nicht ausreichend definiert. Weiters müssen sich verschiedene Akteure auf die bestimmten Belange des Prozesses gezielt fokussieren können. So benötigen die Akteure zur Vermeidung, Erkennung und Behebung von strukturellen Problemen in Geschäftsprozessen wie zum Beispiel Deadlocks einen fokussierten Einblick in die aufgerufenen Datenzugriffsdienste des Prozesses. Wenn jedoch die Anzahl der Prozessaktivitäten und damit die Anzahl der persistenten Datenzugriffsdienste immer mehr ansteigt, kann das Auffinden bestimmter Datenzugriffsdienste des Prozesses eine unmögliche Aufgabe für die Akteure werden.

In dieser Dissertation fokussieren wir uns auf eine bessere Integration persistenter Datenzugriffe in dienstorientierte Geschäftsprozesse. Wir stellen das view-based data modeling framework (VbDMF) vor, das für die Modelling persistenter Datenzugriffe von Prozessen entwickelt wurde. Basierend auf VbDMF, führen wir in das Konzept der persistenten Datenzugriffsflüsse zur Vermeidung, Erkennung und Behebung struktureller Probleme in Prozessen ein. Um die persistenten Datenzugriffe besser warten und wiederverwenden zu können, präsentieren wir einen ganzheitlichen architekturellen Ansatz zur Verwaltung unserer VbDMF Modelle und Modellinstanzen: Eine modellgetriebene Datenzugriffsarchitektur, genannt VMDA. Wir illustrieren die Verwendbarkeit unserer Konzepte in mehreren industriellen Fallstudien. Weiters beweisen wir die Komplexität und Korrektheit der verwendeten Algorithmen. Schließlich belegen wir die Anwendbarkeit der VMDA quantitativ anhand von Messergebnissen.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

BPEL Business Process Execution Language

DAO Data Access Object

DAS Data Access Service

EMF Eclipse Modeling Framework

GIS Geographic Information System

Java EE Java Platform Enterprise Edition

MDA Model-Driven Architecture

MDD Model-Driven Development

ORM Object Relational Mapping

POJO Plain Old Java Object

RDBMS Relational Database Management System

REST REpresentational State Transfer

RPC Remote Procedure Call

SOA Service-Oriented Architecture

UML Unified Modeling Language

URI Uniform Resource Identifier

VbMF View-based Modeling Framework

VbDMF View-based Data Modeling Framework

WFS Web Feature Service

WSDL Web Services Description Language

XMI XML Metadata Interchange

XML Extensible Markup Language

CHAPTER 1

Introduction

1.1 Context

In modern process-driven service oriented architectures (SOAs), process activities can invoke services in order to fulfill business requirements. A service offers a well-defined interface[109], i.e. a web service is a special type of service that provides an interface specified by a web service description language (WSDL)[140]. Service repositories[25,58] manage these services and support service discovery at runtime. Accordingly, process activities can query a service repository dynamically in order to find a suitable service. Besides invoking services, process activities can perform human tasks, do transformations, and/or invoke other process activities[83–85]. In long-running processes, that involve human interactions[143], process data usually need to be read and written data from persistent data storages, typically from relational database management systems (RDBMS)[111] or flat files. Nowadays, this persistent data access is commonly done by so-called data access services (DAS)[20,114,133,146]. DAS are special types of services designed to encapsulate persistent data access details from the service consumer. As common services, the DAS can either be invoked by another services or by process activities directly. In object-oriented environments, DAS commonly use a layer of data access objects (DAOs) to read and write data from physical data storages. The DAO pattern is part of the Java EE pattern catalog[106] and is designed to encapsulate the connection to the data source.

1.2 Problem Statement

A process-driven SOA is an architectural style for developing large business applications[3]. Here, a huge number of processes, processes activities, services, and in particular data access services (DAS) need to be managed. As a result, finding certain DAS for reuse can become an impossible task for stakeholders.

Unfortunately, the relationships between business processes, the DAS, and the underlying persistent data access are not sufficiently elaborated, yet. In service oriented architectures, services can be registered to a service repository. Nowadays, business process execution languages such as BPEL[98] are used as the missing link to incorporate services into business processes[66]. These business process execution languages provide higher level control for services as they describe the services to be invoked and which operations shall be called in what sequence. However, the business process execution languages do not integrate the semantics of an invoked service such as which process activity reads or writes which data. In contrast, they rather regard the process internal data read and written by process activities.

Figure 1.1 overviews the missing links from the user's point of view: The WSDLs stored in *service repositories* are neither associated with the DAS source code from **source code management systems**, nor with the service internal documentation stored in *documentation management systems*, nor with the data storage schemes managed by *data storage management systems*. Accordingly, the service internal documentation of the *documentation management systems* is not connected with the DAS in the *service repositories*, the DAS source code in the **source code management systems**, and the data source schemes in the *data storage management systems*. However, from our experiences, in order to efficiently maintain DAS, a further integration of the DAS, the DAS source code, the DAS documentation, and the data storage schemes is compulsory. In the following we describe the related problems experienced when maintaining, reusing, developing, and tracing a huge number of DAS in a large enterprise in more detail.

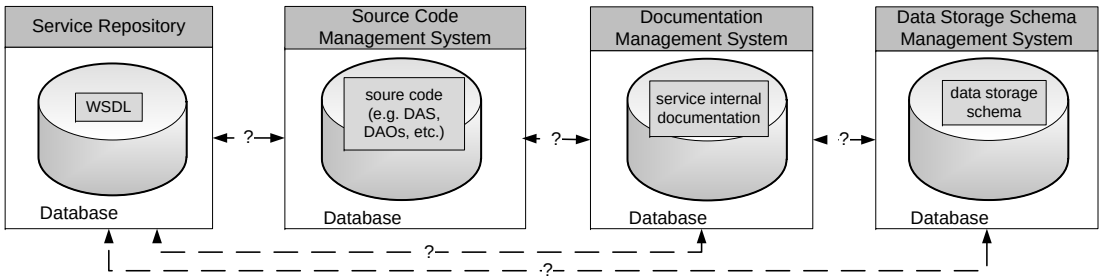


Figure 1.1: Missing Links between Data Access Services, Source Code, Documentation and Data Storage Schemes

Difficult maintainability In organizations, usually data storage schemes are subject to changes. In order to efficiently maintain DAS, it is important to know which DAS are concerned by a change. If a database table schema is redesigned e.g. in case of altering a column, it will be essential to find all DAS that read or write data from this table in order to adapt them. Accordingly, if a table is dropped, some DAS will be obsolete and

should not be available anymore. Due to lacking integration of DAS and the physical data schemes, further elaboration to improve maintainability of DAS is required.

Insufficient reuse of persistent data access best practices DAO implementations use techniques for mapping data objects to physical database tables. These object relational mappings (ORM) have already been subject to extensive research and development. However, in some cases there are several ways in which the mapping can be performed, and the resulting design decisions are typically based on performance or other issues. Moreover, "up-to now attaining good performance still requires careful optimization based on expert knowledge, which can make programs difficult to maintain and evolve"[29]. Thus, in order to improve development productivity, there is a need to reuse these DAOs in particular within teams and departments, but also within the overall enterprise or between partner organizations. Though DAOs are critical components in terms of performance, DAOs are hardly reused. A basic reason for this is that finding a suitable DAO for reuse among hundreds of DAOs usually is a time-consuming task.

Different stakeholders have different requirements Moreover, different stakeholders involved in a business process should be able to understand the SOA only from their perspective. For instance, service developers require mainly information about which DAS access which data, process developers require DAS rather as interfaces to the data, and IT-architects require the big picture of the process/DAS interconnection

Tight coupling between persistent data access and business logic in processes The decision which alternative path to run in the business process often depends on persistent data. Thus, there is a tight coupling between persistent data access and business logic of a business process. This tight coupling is necessary to enable stakeholders to get a basic understanding of the overall process. However, when the number of activities in the process grows, focusing on particular activities of the process flow, such as the process activities reading or writing persistent data (referred to as persistent data access activities throughout this thesis) is a time-consuming task. In data-intensive applications, in order to solve structural problems concerning persistent data access in process flows e.g. deadlocks, stakeholders such as data analysts, DAS developers, and database testers need to overview and analyze the persistent data access activities.

1.3 Motivating Scenario

In the following we motivate the concepts presented in this thesis work. For this, we consider a department of a large organization in which five development teams are each responsible for developing e-government applications. Each team is in charge of planning, analyzing, designing, developing, testing, and maintaining a certain business application through the complete software cycle. In order to access the persistent data in the business applications, each team provides its own data access services (DAS). These DAS are used by the team itself and are consumed by other teams. The DAS implementations can

incorporate Data Access Objects (DAO) encapsulating object-relational mappings between data objects (POJOs) and data storage schemes, and connections to the database. The development teams only need to view their own service implementations and have to search for suitable services in other teams for reuse. On top of these development teams, there is an executive board of a team of IT-architects responsible for the overall architecture. The IT-architects focus on the big picture of the business process persistent data access interconnection. The IT-architects job is to find out technology synergies between the development teams and to consider about system and technology replacements. Thus, they should have an overview about the whole architecture, and should be able to inspect implementation details such as the service provider urls, object-relational mappings, and the database connections. Moreover, in order to achieve improvements such as business application refactoring, they have to know the data dependencies between the teams such that which business applications from one team access certain data from another team.

As the number of process activities grows, the development teams usually can no more survey which services access which persistent data storages. Likewise, the IT-architects typically collect their required information from the development team members or access most likely out-of-date documentation.

In this thesis, we fill the gap between processes and their DAS. As a result, both the development teams and the IT-architects can have an overview of the overall architecture including service implementation details such as the database connections of a business application.

1.4 Research Questions

Research Question 1: How to improve documentation of the relationships between processes, services, and persistent data access? Stakeholders need to understand the relationships between process activities, the invoked DAS, the DAOs, and the physical data storage schemes. E.g. persistent data access documentation should contain which database tables relate to which data objects and which data objects are used by a DAS. Unfortunately, documentation approaches to trace these relationships usually lack quality. Accordingly, most software engineers do not update most software documentation in a timely manner. "The only notable exception is documentation types that are highly structured and easy to maintain, such as test cases and inline comments"[77]. This research question focuses on how to make persistent data access in process-driven SOAs easily understandable and documentable.

Research Question 2: How to support different stakeholders to model and get their relevant part of persistent data access? Different stakeholders involved in a business process should be able to understand persistent data access from their perspective. For instance, data analysts require mainly information about which DAS access which persistent data, business process developers require DAS rather as interfaces to the data, and database administrators focus on the physical data storage schemes and

database connection details. This research question focuses on how to make persistent data access in process-driven SOAs easily accessible to different stakeholders.

Research Question 3: How to support stakeholders in solving structural problems in business processes? Unfortunately, the process activities are usually tightly coupled. Thus, when the number of activities in the process grows, focusing on particular activities of the flow such as the persistent data access activities invoking DAS is a time-consuming task. In data-intensive applications, in order to solve structural problems in business processes such as deadlocks, stakeholders need to overview and analyze the persistent data access activities of the process flow. This research question deals with how to support stakeholders in solving structural problems concerning persistent data access in business processes.

Research Question 4: How to build an architecture to enhance manageability, maintainability, and reuse of persistent data access in a SOA? Though DAS are critical components in terms of performance, they are hardly reused. However, in order to improve development productivity, there is a need to reuse DAS in particular within teams and departments, but also within the overall enterprise or between partner organizations. Moreover, in order to efficiently maintain DAS, it is important to know which DAS are concerned by an underlying change. In example, if a physical storage schema is redesigned e.g. in case of altering a column, it will be essential to find all DAS that read or write data from this table in order to adapt them. This research question focuses on the requirements decisions for enhancing manageability, maintainability and reuse of DAS.

1.5 Scientific Contributions

In order to answer the research questions before, this thesis achieves a series of scientific contributions as specified as follows. Figure 1.2 gives an overview of these contributions. In the figure, the number, dedicated to each of the contributions, corresponds to the number of the contribution, described below. The bars on the left and on the right in the figure denote the outcomes of the contributions. Accordingly, both contribution 1 and contribution 2 aim at improving *documentation* and *traceability* of persistent data access in process-driven SOAs. Furthermore, a better *stakeholder support* results from the 1st contribution, whereas the 2nd contribution focuses on structural *problem solving* in business processes. Moreover, with both contribution 3 and contribution 4, we can achieve better *maintainability* and *reusability* of persistent data access in process-driven SOAs.

1. **View-based data modeling framework (VbDMF)** Based on the concept of separation of concerns and the view-based modeling framework (VbMF) we developed a set of data-related models tailored to the requirements of different stakeholders. The data-related extension of VbMF, the view-based data modeling

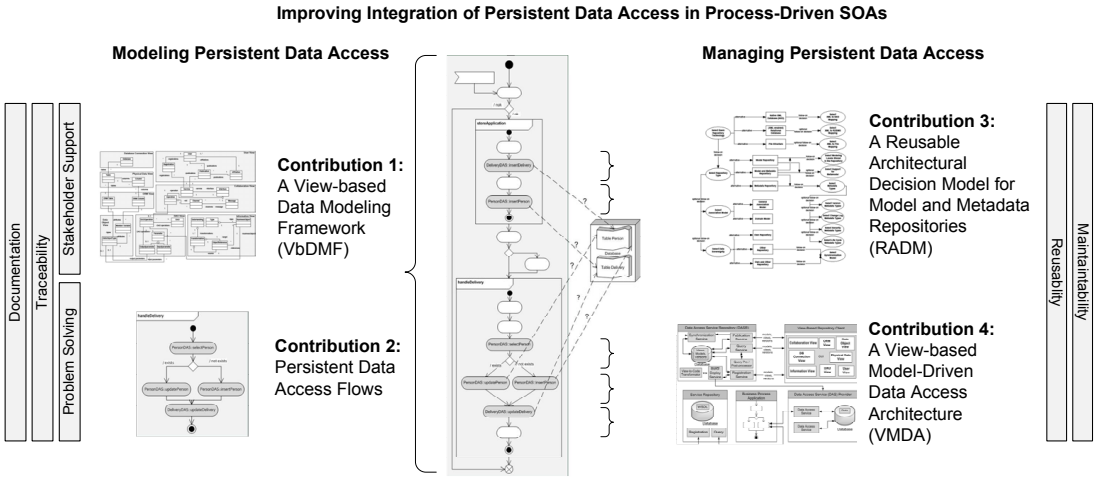


Figure 1.2: Contributions Overview

framework (VbDMF), introduces a layered model for accessing data in process-driven SOAs[83]. These models are used throughout this thesis to improve traceability, documentation, and maintainability of persistent data access in process-driven SOAs. Consequently, these models have been specified to answer research question 1 and 2. VbDMF contributes to increase integrity of DAS within the process-driven SOA by connecting process activities with DAS and the DAS with their underlying implementation such as the DAOs, object-relational mappings, physical data storages, and database connections. Different data-related stakeholders such as data analysts, DAS developers, database testers, etc. can view data-related information according to their particular interest.

2. **Persistent data access flows** In order to answer research question 3, we applied the concept of persistent data access flows to solve structural errors in business processes. In this thesis, we introduce tailored persistent data access flows to support stakeholders in solving structural errors in business processes. Several motivating scenarios illustrate how different stakeholders can use our persistent data access flow concept to solve different kinds of problems. A view-based model-driven solution validates the feasibility and applicability of the approach. Finally, we prove the correctness and the complexity of the used algorithms.

3. **A reusable architectural decision model (RADM) for model and metadata repositories** In order to answer research question 4, we firstly provide architectural decision-support for architects in planning and setting up model and metadata repositories. With our RADM we stakeholders can find solutions in resolving fundamental design problems arising during the design phase. For each decision we present recommendations which alternative to choose depending on

certain requirements and boundary conditions. Hence, given a set of individual requirements, our architectural decision model helps in setting-up a repository, as well as building a custom repository. In a case study we illustrate the major design decisions made when setting-up our own Data Access Service (DAS) repository.

4. **A view-based model-driven data access architecture (VMDA)** Our VMDA focuses on better managing DAS in process-driven SOAs, and thus, contributes to increase DAS reuse and maintainability. In order to solve research question 4, our novel contribution combines four basic concepts (DAS, MDD, DAS repository, VbMF/ VbDMF). To fulfill the requirements such as dynamic changes of data sources in a process-driven SOA, we support DAS to read and write data from a data source. However our architecture approach can be reused to integrate other types of persistent data access implementations into DAS. We use the model-driven development (MDD) [139] approach to be able to abstract DAS from a higher level than the source code layer. Another useful development aspects of MDD, we can make use of, are automatic source code generation and deployment. To provide management support for services, we introduce a DAS repository storing DAS models and model instances. In particular, our DAS repository architecture provides a query service with which remote repository clients can discover models and model instances stored in the central repository. Our model-driven solution is based on VbMF and VbDMF. The concept of separation of concerns contributes to enhance documentation of processes, DAS, DAOs, the underlying physical data storage schemes, and the relationships between them in process-driven SOAs.

1.6 Previously Published Work

Conference papers:

- We have specified the View-based Data Modeling Framework (VbDMF) for modeling persistent data access in process-driven SOAs. VbDMF and the fundamental concepts of this thesis were published in the following publication. We presented the VbDMF fundamental concepts at the ServiceWave conference in Madrid. C. Mayr, U. Zdun, and S. Dustdar. Model-Driven Integration and Management of Data Access Objects in Process-Driven SOAs. In P. Mähönen, K. Pohl, and T. Priol, editors, ServiceWave, volume 5377 of Lecture Notes in Computer Science, pages 159–170. Springer, 2008.

- The reusable architectural decisions for setting-up model and metadata repositories were published in the following paper: C. Mayr, U. Zdun, and S. Dustdar. Reusable Architectural Decision Model for Model and Metadata repositories. In FMCO, pages 1–20, 2008.

Journal papers:

- Our view-based model-driven architecture for enhancing maintainability of data access services was published in the following journal: C. Mayr, U. Zdun, and S. Dustdar. View-based Model-Driven Architecture for Enhancing Maintainability of Data Access Services. In Data & Knowledge Engineering, September, 2011.

- Our approach to improve traceability of persistent data access flows in process-driven SOAs is to be published in the following journal: C. Mayr, U. Zdun, and S. Dustdar. Improving Traceability of Persistent Data Access Flows in Process-Driven SOAs. In Distributed and Parallel Databases (under review).

1.7 Thesis Structure

The rest of this thesis is organized as following.

Chapter 2 introduces the state-of-the-art in managing and modeling persistent data access in process-driven SOAs. Furthermore, we compare our contributions to related work and explain how our contributions emphasize from this related work.

Chapter 3 introduces the background concepts of this thesis to better understand our contributions. These are service-oriented architectures, model-driven development, persistent data access, and the view-based modeling framework.

Our four major contributions are split into modeling and managing persistent data access in process driven SOAs. The following two chapter focus on modeling persistent data access in process-driven SOAs:

Chapter 4 introduces the View-based Data Modeling Framework (VbDMF) with a set of data models used to specify the relationships between process, service, data object, and persistent data storage. In this thesis we use VbDMF to prove our concepts aiming at a better management of persistent data access in process-driven SOAs.

Chapter 5 addresses the tight coupling of process activities by extracting tailored persistent data access flows from the whole process flow. To the best of our knowledge persistent data access flows are not used to solve structural problems concerning persistent data access in business processes, yet. In a series of motivating scenarios we illustrate how the persistent data access flow concept can be applied to problem solving. Moreover, we qualitatively and quantitatively prove both the applicability and feasibility of our approach.

The following two chapter focus on managing persistent data access in process-driven SOAs:

Chapter 6 describes reusable knowledge in form of reusable architectural decisions for IT-architects in setting-up, planning, and developing model and metadata repositories. A case study illustrates the decisions made when setting up our own data access object model repository by walking through the reusable architectural decision model.

Chapter 7 focuses on bridging the gap between the DAS and their implementation by presenting a view-based, model-driven data access architecture (VMDA) managing models of the DAS, DAOs and database queries in a queryable manner. Furthermore, we describe tool-support, and illustrate the applicability of our VMDA in an industrial, large-scale case study. By evaluation, we quantitatively prove that our approach performs with acceptable response times.

Finally,

Chapter 8 summarizes our research problems and the major contributions, and presents some potential future work.

CHAPTER 2

Related Work

In this chapter we present related work from the existing literature, related standards, and known uses. We also emphasize the contribution of our work by explaining how our work compares to these related works. As our thesis is about managing and modeling persistent data access in process-driven SOAs, this chapter is organized as follows: In Section 2.1 we present the related work concerning managing persistent data access in process-driven SOAs. Finally, in Section 2.2 we relate our View-based Data Modeling Framework (VbDMF) to alternative solutions modeling persistent data access in process-driven SOAs.

2.1 Managing Persistent Data Access in Process-Driven SOAs

To be able to manage persistent data access in process-driven SOAs, we were inspired of repositories in general. In[14], Bernstein and Dayal give a fundamental overview of repository technology as well as functional requirements of a repository. We compare our architectural concepts with common service and model repositories. Afterwards we relate our VMDA to alternative DAS architecture approaches. Furthermore, we contrast the VMDA with other view-based systems. Finally, we check our model-driven approach against the field of semantic knowledge discovery.

Service repositories Our work is in particular influenced by repositories incorporating metadata. A common representative of metadata repositories are service repositories. These service repositories contain metadata about location information such as service bindings according to the web service description language (WSDL). There exist several web service registry standards and implementations. Common standards are UDDI[25], ebXML[99], and WSIL[18]. Examples of related implementations are the ebXML

repository reference implementation[43] and the WebSphere service registry and repository (WSRR)[58] that is based on UDDI. Like our view-based model-driven data access architecture (VMDA), ebXML web service registries[99] have interfaces that enable submission, query, and retrieval of the contents of the registry. However, standards such as UDDI are not adequate enough for finding suitable services[5]. In example, due to missing key words and unsatisfactory documentation retrieving DAS is often impossible. Consequently, there is a need to utilize service context during the discovery process[125]. In our approach, we can search for DAS by more sophisticated criteria. Known information about the underlying databases, tables, columns, and ORM frameworks can be exploited for a more targeted DAS search and thus enable us to achieve better search results in less time. In order to integrate our DAS repository into process-driven SOAs, we adopted the basic CRUD interface abstractions, used in these approaches, and assemble them into our DAS repository architecture. Moreover, these service repositories such as UDDI[25] strictly separate the interfaces from their implementation. In contrast to these approaches, our VMDA integrates the DAS with the underlying DAOs, object-relational mappings, and database connections, and we can thus provide a high-quality documentation of currently available and deployed DAS. This documentation can comprise both the DAS, contingently underlying DAOs encapsulating the database queries, as well as the data storage schemes.

Model repositories There are a series of model repository standards and implementations (e.g.[89],[95]). An interesting approach is the one of Milanovic et al.[89] who present the design and implementation of a repository that supports storing and managing of various artifacts such as meta-models, models, constraints, metadata, specifications, etc. They illustrate the repository's data model specifying the stored artifacts and artifact metadata such as versioning information. Furthermore, they give an overview of the repository architecture, and describe how to manage artifacts from the repository's point of view. However, they do neither specify client-server interactions nor how to synchronize with other repositories. Furthermore they do not provide an overview about different types of metadata such as those presented in our work. They exemplary illustrate the design of the BIZYCLE repository architecture without identifying architecture decisions to select different alternatives and options. In this thesis, we also describe the basic repository services from a user's point of view. Instead of involving management issues such as project management and user control, our decisions primarily deal with the question which artifacts shall be stored in a repository and how to model the associations between them.

Nissen and Jarke's encouraging work propose repository support for goal-oriented inconsistency management in customizable multi-perspective modeling environments[95]. Their repository approach aims at integrating meta-meta-models, meta-models, conceptual models, and model instances. Thus, Nissen and Jarke focus on the relationships between the different modeling levels. Like their approach, our repository approach stores meta-models, models and model instances and manages theses artifacts. However, in contrast to creating new perspectives for each modeling level, we concentrate on

creating views within a specific modeling level in order to enable stakeholders to focus on several concerns of the overall model instance. According to the concept of separation of concerns, in our work, database administrators can focus on the Physical Data View describing database tables whereas DAO developers can focus on describing DAOs of the DAO View.

Min et al.[90] present an XML data management system using a relational database as a repository that translates XQuery expressions into a single SQL statement. They provide powerful searching using the standard XQuery language. However, e.g. in order to create source code from the defined models, the models are usually based on a meta-model. Hence, if the models contain embedded metadata elements, then using XQuery to query the model elements will be very complex for stakeholders who do not know the underlying meta-model. On the contrary, we use a lightweight easy-to-learn language based on key word search conditions which fulfills the requirement to search for view model instances and view models by different search criteria. Thus, our query language supports user-friendly key word search in XMI model instances and models whereas Min et. al. 's approach lacks usability, when stakeholders only have a limited knowledge of the metadata within the XML models.

France et.al.'s interesting approach[42] introduces a development plan for setting up model repositories storing MDD artifacts. In contrast to our approach, the authors of the ReMoDD project in particular focus on the types of interactions that are most useful for repository users. Besides, the ReMoDD project's scope of research does not include storing metadata.

Finally, there are many articles that focus on each of the architectural decisions for setting up model and metadata repositories for their own. For example, several work[10,52] focus on algorithms of mapping XML model instances to a certain repository storage type. However, for the best of our knowledge there is no work that connects all these illustrated architecture decisions with each other.

Data Access Service Architecture Approaches The most related to our work is probably the architectural approach of Zhu et al.[146]. Like our approach, they use data access services (DAS) to address the problem of large scale data integration where the data sources are unknown at design time. More specifically, their architecture approach proposes an integration broker service in order to establish a high level integration of DAS into the SOA. Likewise, they focus on semantic description and discovery of DAS. However, they do not describe how these semantic descriptions are linked with the DAS. In contrast, in this thesis, we propose a model-driven view-based approach to describe these semantic descriptions used to discover DAS in a SOA. Whereas their approach specifies a high-level architecture, we rather present a continuous approach to develop, maintain and manage DAS.

Like our approach, Resende uses DAS to manage heterogeneous data sources in SOA environments[114]. Further, he describes how to efficiently handle persistent data access with service data objects (SDO)[114]. Like our approach, Resende uses DAS to access the data. In contrast, in their solution, the DAS are based on the SDO programming model

in order to handle data across heterogeneous data sources fit for a SOA environment. In our approach we use the DAS as a general abstraction layer for integrating data into the SOA rather than defining a certain implementation technology of the DAS. Accordingly, our approach is more general, because the DAS can be implemented on top of various service technologies such as SDO or the Java Architecture for XML Binding (JAXB) [136] to transform XML data formats into objects of object-oriented programming languages. This unmarshalling is encapsulated by the DAS used for uniformly accessing heterogeneous data sources.

View-based approaches To the best of our knowledge, up to now there is no work that explicitly proposes a view-based model-driven architecture for managing and maintaining DAS. However, there are many approaches using views in order to enhance maintainability and traceability. In particular, our view-based concept is similar to concerned-based and multi-perspective software development approaches:

Robillard et al. [117] present a system called ConcernMapper in order to enable a simple view-based separation of scattered concerns. The basic idea of ConcernMapper is to allow developers to associate parts of a program with high-level concerns. Their approach supports developers in development and maintenance tasks involving scattered concerns by allowing them to organize and view the code of a project in terms of high-level abstractions called concerns. Like our approach, an extensible platform is intended to provide a simple way to store and query concern models. However, the concerns are only subject to developers, whereas our view-based models can be tailored to the requirements of diverse stakeholders. Moreover, concerns can only be retrieved, when the relevant source code section has been mapped to concerns, before. Due to our model-driven approach, we can retrieve all model elements and their relationships by diverse search criteria.

Nuseibeh et al. [97] apply their viewpoints framework to focus on method engineering in a multi-perspective software development environment. In order to manage the diversity in composite systems, their viewpoints, like our views, are defined to be loosely coupled and encapsulate specific knowledge about processes,systems, and domains. Moreover, as VbDMF, their viewpoints are described and developed following a formal notation and development strategy. In contrast to our approach, their view-based solution is an organizational framework. Our DAS repository is rather designed to store loosely coupled models that have defined connection points to support a flexible integration within the models. In addition, due to our model-driven approach, our model instances can be transformed to other outputs such as source code and documentation.

Semantic knowledge discovery There are several approaches for semantic knowledge discovery e.g. [122], [20]. Representatively, we refer to the software architecture of Cannataro et al. [20] for distributed knowledge discovery. The paper discusses how the knowledge grid can be used to implement distributed data mining services. These data mining services are defined to search, select, extract, and transform data from specific data sources. In particular, data sources can be found based on user requirements and

constraints. The disadvantage of these semantic approaches is that "the semantic service discovery is more time-consuming due to the additional context and semantic matching modules"[80]. In this chapter, we show that our query engine performs much better than these semantic discovery approaches. In our view-based model-driven data access architecture (VMDA), we store structured model instances. Thus, with our VMDA, there is no need to extract structured data from plain text by semantic services. Moreover, we can reuse the high-structured DAS for model-to-code and model-to-documentation transformations.

2.2 Modeling Persistent Data Access In Process-Driven SOAs

In this section, we relate our view-based model-driven solution to related work concerning modeling persistent data access in process-driven service-oriented environments. Hereto, firstly, we compare to various approaches focusing on better integrating persistent data into the overall SOA[22,48,141,144]. Secondly, we compare our model-driven solution to existing business process modeling systems (BPMS). Finally, we contrast our VbDMF with alternative solutions concerning solving structural problems in business processes.

Modeling persistent data access

In the literature, various related works propose using data access services (DAS) for better data integration.

Carey et al.[22] examined how the AquaLogic Data Services Platform (ALDSP) supports data modeling and design. They describe the ALDSP 3.0 data service model and assert that the modeling extensions in ALDSP 3.0 provide a rich basis for modeling data services for SOA applications.

Wang et al.[141] propose a dynamic data integration model architecture based on SOA. On the basis of XML technology and web service, their architecture model enables data sharing and integration over all business systems. Thus data resource and information interoperability is realized in a cross-platform manner.

As our approach, both ALDSP[22] and the dynamic data integration model[141] use data access services to read and write data. However, they focus on separate modeling of data access services for use in external environments. In these two approaches, data integration overall business systems is established by using DAS as interface to the data. In contrast, we propose a continuous integration approach to be able to exploit the structured nature of the data access service models in business processes.

Zhang et al.[144] propose a new process data relationship model (PDRM) to specify the complex relationships among process models, data models, and persistent data access flows. In our approach, we define the activities incorporating data access as DAS activities. Likewise, Zhang et al. define these activities as data access nodes (DAN). Like our approach, they understand data access flows as persistent data access flows rather than data flows representing transient and persistent data. However, as opposed to our

approach, they focus on automatic data access component generation to cluster similar data access components into larger components. In contrast, we concentrate both on the applicability and the feasibility of generating data access flows. Furthermore, unlike our model-driven solution, they solely focus on simple activities and cannot model structured process activities.

Zhang et al.[48] introduce a unique information liquidity meta-model (ILM) to separate persistent data integration logics from business services and application services. Like our approach their architecture uses a data service layer to access the data. In addition, as our approach, they use views to relate processes to new and existing services, or new and existing data definition. Whereas our approach focuses on modeling new view models and extracting new view models from existing view models, they solely concentrate on creating new models. In our approach we also focus on solving data analysis problems by creating flattened persistent data access flows from the whole process flow.

Developmental related work Nowadays, there still is a missing link between the programming components and the data storage schemes[83]. This gap results in several object-relational mapping (ORM) problems[41]. A straightforward approach to solve this problem are cartridges, such as those provided by AndroMDA[7] or Fornax[39]. Cartridges support separation of concerns by providing mechanisms for accessing and manipulating data through DAOs. They are predefined components for model-driven generators that enable developers to integrate DAOs into services by generating either an instance of a DAO interface into the service code[7] or generating DAO instances into the service operation[39]. However, the relationships between DAO operations and service operations are not specified so far by cartridges. Even though the Fornax cartridge[39] connects DAOs to service operations, it lacks information about which DAO operations are invoked by which service operation. This information, however, is important for stakeholders, such as software architects and service developers, to gain a clear view about which database transactions are invoked by which service operation. To overcome this problem, we extend the cartridge approach with the integration of DAO operations into DAS operations.

Finally, SVN/CVS version management systems are closely related to our approach. These systems can act as a DAS repository by historicizing all versions of DAS/DAO model instances. However, this approach has some major limitations: When developers want to reuse committed source code from the version management system, they have to check-out the specific components, provided that they know exact names (or at least roughly the names) of the components that should be reused. However, if they do not know the exact name of a component to be reused, all DAS/DAOs will have to be checked-out from the version management system. Hence, a local full text search is necessary to find DAS/DAOs by a key word, e.g., by a column of a database table. In contrast, our DAS repository provides searching mechanisms to retrieve suitable DAS/DAOs by diverse search criteria with acceptable response time.

Business Process Modeling Systems (BPMS)

Our work is closely related to common commercial and open-source business process modeling systems (BPMS)[142]. Representatives of common commercial BPMS are IBM Websphere MQ Workflow[59], Webmethods[123], and TIBCO[130]. In addition, there are common representatives of open-source systems i.e. JBOSS[64] and Intalio[60].
Russel et al.[118] define a specific data interaction pattern for how BPMS access persistent data. Their main focus is to determine data patterns in business processes. On top of this data integration pattern, our conceptual approach focuses more on solving structural problems in business processes by using persistent data access flows. Unfortunately, many BPMS do not explicitly support this pattern by a direct integration of persistent data access into the process activities. In example, IBM Websphere MQ Workflow[59] and Intalio BPMS Designer[60] do not provide an explicit mechanism to invoke persistent data access from the process activities within the BPMS. In these BPMS, the persistent data is rather accessed e.g. through underlying services incorporating the persistent data access implementation.

Other BPMS such as Webmethods[123], JBOSS Messaging[63] and TIBCO[130] support integration of persistent data access into the process activities. In these BPMS, process activities can directly request persistent data within the BPMS environment. However, as opposed to our persistent data access flow concept, these BPMS do not provide comparable support to adequately overview persistent data access in data-intensive business processes. In Webmethods[123], stakeholders can configure adapter services used to read or write data from the database, in example the InsertSQL, UpdateSQL and DeleteSQL services. As our approach, the services can configure SQL statements in a structured way. For example, statements can contain structured elements such as database connection properties, database tables, and database table columns. Furthermore, Webmethods provides filtering mechanisms to limit the adapter services by structured elements such as catalogs, schemes, and tables. JBOSS Messaging[63] supports configuration of relational database connections by the JDBC Persistent Manager. A channel mapper is used to configure SQL statements such as *Create* and *Select*. However, as opposed to our approach, JBOSS Messaging does not support structured modeling of persistent data access. In TIBCO[130], it is possible to establish the link between a process activity and structured process data models with business objects, e.g. specified in UML[74], designed with TIBCO Business Studio. TIBCO provides tooling support to read/write access from the business objects.

Solving Structural Problems in Business Processes

There are several approaches concerning solving structural problems in business processes[11,119]. Sadiq et al.[119] identify structural conflicts in process models by applying graph reduction rules. Awad et al.[11] use business process queries to detect structural problems in business processes. In contrast to these works, in our approach, besides solving structural problems in business processes, we aim at enhancing traceability and documentation of persistent data access. Moreover, we provide a model-driven solution

to reduce business process complexity as we can flatten business processes by certain filter criteria.

At this point we want to clearly differentiate our approach from other works such as BPELDT[49] which focuses on the data flows transporting data from one process activity to the next process activity. Habich et al. introduce BPEL data transitions to efficiently model "data-grey-box web services"[49]. In contrast, in our approach, we focus on tracing the persistent data access activities themselves instead of the transitions between two process activities.

Static analyzing techniques There are a number of frameworks for performing static analysis to extract common data flows from the whole program. An example of these static analysis approaches is the demand-driven flow analysis as proposed by Duesterwald et al.[31]. "The goal of demand-driven analysis is to reduce the time and space overhead of conventional exhaustive analysis by avoiding the collection of information that is not needed"[76]. As our approach, Duesterwald et al. focus on extracting sub flows from process flows on-demand. However, they aim at extracting common data flows instead of persistent data access flows. Moreover, we provide a model-driven, view-based approach to analyze and document persistent data access flows in process-driven SOAs.

Testing There are a number of approaches in the literature that elaborate on test case creation and selection.

Fischer et al.[37] focus on improving test case quality for declarative programs by introducing a novel notion of data flow coverage. In their opinion, a visual representation of the control- and/or data flow would help the users to better understand program execution. We share the opinion that a visual view increases the understandability of the data flows. However, our views are not restricted to the viewing of these data flows. Accordingly, our persistent DAS Flow View can be integrated with other views in order to form richer views, in example with the DAO View, the ORM View, and the Physical Data View.

There are several works using data flows for selecting test cases as presented in[112]. Rapps et al. apply data flow analysis techniques to examine test data selection criteria. The procedure presented associates each definition of a variable with each of its usages within a flow. The data flow criteria that they have defined can be used to traverse each path. Like our approach, each persistent data access activity in the process flow can be associated with corresponding definitions. In contrast to this formal approach, we use a visual approach for selecting our test cases.

Deadlock detection and prevention In the following we relate our solution to various static and dynamic deadlock detection techniques.

There are many runtime approaches (such as[61] and[113]) that aim at deadlock-free sharing of resources in distributed database systems. Isloor et al.[61] distinguish between deadlock detection, deadlock prevention and deadlock avoidance techniques. Krishna et al.[113] present a graph-based deadlock prevention algorithm that reduces processing

delays within the distributed environment. However, with our persistent data access flow approach we provide a visual solution in order to discover errors at the earliest stage of development – at the modeling level.

There are a number of static deadlock detection algorithms (e.g.[92]). Naik et al's[92] deadlock detection algorithm uses static analyses to approximate necessary conditions for deadlocks to occur. Their effective algorithm concentrates on detecting deadlocks between two threads and two locks.

Dedene et al.[28] present a formal approach to detect deadlocks at the conceptual level. In their work, they present a formal process algebra to verify conceptual schemes for deadlocks based on the object-oriented analysis (OOA) method M.E.R.O.DE. As our approach, Dedene et al. can check the models for deadlocks at the earliest stage in the development process.

An interesting approach is presented by Zhou et al.[145]. Like our approach, the authors use a static approach to analyze deadlocks in data flows. They in particular concentrate on analyzing deadlocks in loops. In order to determine deadlocks, they define a causality interface that abstractly represents causality of data flow actors.

In contrast to these formal deadlock analysis approaches, again, our approach is a visual solution for detecting deadlocks. Furthermore, on top of our approach, common deadlock detection and prevention techniques as described before can be performed. Furthermore our persistent data access flow concept aims at documenting the persistent data access flows within the control flow. Thus, with our approach we do not solely focus on detecting deadlocks in process flows, we rather enable a more general analysis of a series of development and testing problems.

Transaction handling (BPMS) In order to solve structural problems in business processes, common BPMS such as IBM Websphere MQ Workflow[59], JBOSS[64], and Intalio[60] support transaction handling. Thus, in case of failures, the transactions can be rolled back or compensated. Whenever some actions cannot be rolled back e.g. due to external dependencies, a compensation handler can be invoked to perform an "undo action". In contrast, our persistent data access flow approach focuses on the underlying structural problem. Moreover, we solve the cause of the failed transactions instead of solely handle the problem. In example, if a database table is locked, due to a structural problem in the business process, our persistent data access flow approach will contribute to solve the problem more quickly. In addition, our model-driven provides up-to-date documentation of persistent data access in business processes.

CHAPTER 3

Background

In this chapter we present some background information in order to better understand the contributions in the following chapters.

3.1 Service-Oriented Architecture

Services are self-describing and self-contained components designed to support the compositional development of distributed systems[109]. In a SOA, organizations publish services by suppling their services descriptions and by providing corresponding technical support. Service descriptions include both functional and non-functional specifications of the services. Functional specifications include the service interface descriptions, expected results, and behavior. Non-functional descriptions are Quality of Service (QoS) attributes such as response time, availability, security, scalability etc.[109]. Whereas service providing organizations publish the service descriptions, the service implementation details remain hidden to the service consuming parties. Service consuming organizations can query the state of a service by standard interfaces[109]. Service registries facilitate service providing organizations to describe their services. Similarly, service consumers are assisted in finding service descriptions registered by service providers[71,108]. After locating a service, service consumers can bind the service in order to communicate with it. Figure 3.1 illustrates these relationships between a service consumer, a service provider and a service registry in a SOA.

A web service is a special kind of service, whose service interface descriptions are specified by a web services definition language (WSDL). Web services commonly communicate with each other via SOAP messages (in XML format). Web service consumers utilize the universal description, discovery, and integration (UDDI) standard to locate service providers and discover web service descriptions[26,109]. As a result, Web service consumers can dynamically bind web service descriptions to web service implementations.

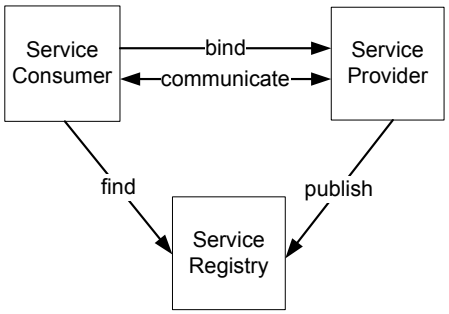


Figure 3.1: Service-Oriented Architecture (SOA) Triangle

The WS-* stack is a collection of protocols and standards, mostly based on HTTP, designed for building web services. This WS-* stack includes WS-Notification, WS-Security, WSDL, and SOAP. In practice, these protocols and standards are used to implement remote procedure call (RPC) applications over HTTP[115]. RPC-style web service clients send an envelope containing the whole set of arbitrary service operations to the service. RPC-style services are accessible via a single service endpoint URI. A common light-weight alternative to RPC-style web services, are REST-style services: REST is an architectural style to manipulate resources using a uniform set of HTTP methods. As REST-style web services are resource-oriented, they expose one URI for each resource of the service[56,108,115]. In our prototype implementation, our models specify RPC-style web services.

3.2 Model-Driven Development

Models are gaining importance in software development, for instance in the model-driven development (MDD)[46,139] field, as well as in other disciplines such as biology and physics. Today many systems are modeled with precisely specified and detailed models. Reasons are among others the increasing support for model interoperability between modeling tools[116] and the increasing use of model-driven development (MDD)[46,139]. In MDD many tools in a tool chain must work on a set of models, and they must be able to import models developed with external modeling tools. Model repositories[32,72,124] support this trend by managing modeling artifacts, such as models, model instances, model relationships, and so on. A model repository enables modelers to create, retrieve, update, and delete modeling artifacts, and to query for them. Usually additional metadata about the modeling artifacts can be stored and used in the queries. These model and metadata repositories can support extra functionality, such as versioning support, security functions, or storing of related source code artifacts. Some repositories are even pure metadata repositories.

Repository, Metadata Repository, and Model Repository

The field of repositories is currently a popular area of research. Therefore the following definitions are not exhaustive with regard to a full functional and non-functional requirements specification of a repository. These nominal provisions rather point out those characteristics of a repository we in particular focus on in this thesis.

We define a *repository* as a centrally accessible component storing information about reusable artifacts[14]. Examples of these artifacts are source code, documents, and special-purpose models such as models for defining data objects in object-oriented environments, models for MDD[139], biology models[78], and so on. Furthermore, a repository has to provide the means to query these information artifacts and metadata about these information artifacts respectively according to certain search criteria. In many cases, querying is performed using some query language.

When setting-up a repository, architects can choose between two alternatives. The repository can either provide this information by storing the artifacts themselves, or it stores metadata about where and how a specific artifacts can be accessed, reached, or invoked. We refer to a repository that stores arbitrary or user-defined metadata on artifacts as a *metadata repository*. Typical examples of (categorized) information, metadata repositories use, is information about users, versioning, affiliations, etc.

When a repository provides models and/or model instances such that it either stores models and/or model instances as its artifacts or provides these models and/or model instances stored at other locations, we refer to a repository as a *model repository*.

Usually, a model repository additionally provides metadata of models or model instances. Hence, we refer to a repository that provides metadata of models and/or model instances as a *model and metadata repository*.

3.3 Persistent Data Access

Data access services (DAS) Most service-oriented applications require reading or writing data from a central physical storage, typically an RDBMS. Nowadays, this data access is done by so-called data access services (DAS). DAS are variations of the ordinary service concept: They are more data-intensive and are designed to encapsulate persistent data access as a service[133]. Like a common service consists of service operations, a DAS consists of DAS operations.

DAS can be complemented with many other technologies and concepts such as Service Data Objects (SDO). SDO is a language-independent, unified programming model defining CRUD operations for handling data access across various data sources such as RDBMS, XML, flat files, etc. in distributed systems[114]. DAS can provide a layer of abstraction between the SDOs and the data sources. This abstraction layer enables different SDOs to transparently access heterogeneous data sources[114].

Data access objects (DAOs) In object-oriented programming environments, the DAS can use the Java EE Data Access Object (DAO) pattern[106] to encapsulate access

to the data source. DAOs are a special kind of objects providing access to data that is usually read or written from one or more database tables. The goal of this design is to enhance software maintainability and strict separation of the layers providing business functionality and persistent data access in a SOA. Hence, the DAO hides database-dependent (e.g. SQL queries[87]) and technology-specific (e.g. JDBC[27]) connection details from the DAS provider. Our approach uses DAS/ DAOs as example implementation.

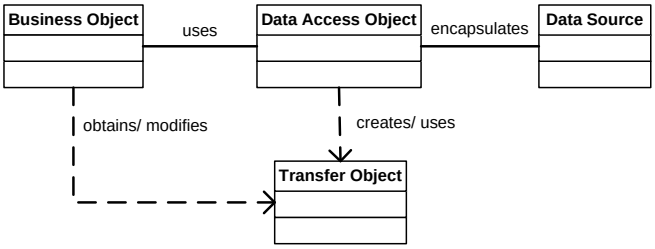


Figure 3.2: Java EE Data Access Object (DAO) Pattern[106]

Figure 3.2 shows a UML class diagram representing the relationships of the Java EE DAO pattern[106]. A *BusinessObject* accesses the data source via the a *DataAccessObject*. The *DataAccessObject* enables transparent access to the data source by encapsulating the underlying data access implementation. A data source can be any kind of physical data storage, typically a relational database management system (RDBMS), but also an XML repository, or a flat file system. If the data source is a service or another system, the *DataAccessObject* can use a *TransferObject* to manipulate or query data from the data source.

Object-relational mapping (ORM) Developers in the field of object-oriented applications use data objects and attributes to describe real-life objects. In order to connect to a relational database management system (RDBMS), they can use a programming API such as JDBC for database-independent connectivity. For querying and manipulation of data, developers typically use SQL, the most accepted and implemented interface language for RDBMS[87]. In a relational database management system (RDBMS), data is stored in database tables and columns. Thus, when object-oriented applications read or write data from an RDBMS, developers need to bridge the gap between two different programming paradigms: They are faced with the problem of how to map objects to tables and reversely[67]. Mapping one data object with a set of attributes to a certain database table with certain table columns is comparably easy. However, the more sophisticated object-oriented programming concepts such as aggregation, inheritance, polymorphism, association between classes, and differing data types need to be implemented by the developers, too.

ORM frameworks such as Hibernate[55] and Ibatis[57] support developers in mapping data between object-oriented programming languages and RDBMS. Due to these tools,

development time can be significantly reduced comparably with manual data handling with SQL and JDBC[126].

3.4 View-Based Modeling Framework

The View-based Modeling Framework (VbMF) defines basic processes in process-driven SOAs. By VbMF different concerns in a business process can be separated into different views. According to the principle of separation of concerns, VbMF enables stakeholders to understand each view on its own, without having to look at other concerns, and thereby reduces the development complexity[132]. By VbMF, view models can be specified at different abstraction levels e.g. more abstract, technology-independent views are separated from technology-dependent views: For example, business process developers, that need to overview the business process and its sub processes, typically use abstract, technology-independent views. Examples of abstract views are the Flow View, the Collaboration View, and the Information View (see Figure 3.3). In contrast, technical experts, such as system architects, are also interested in technical details such as service endpoints or database connection properties. Examples of technology-dependent views are the BPEL Flow View, the BPEL Collaboration View, and the BPEL Information View[132].

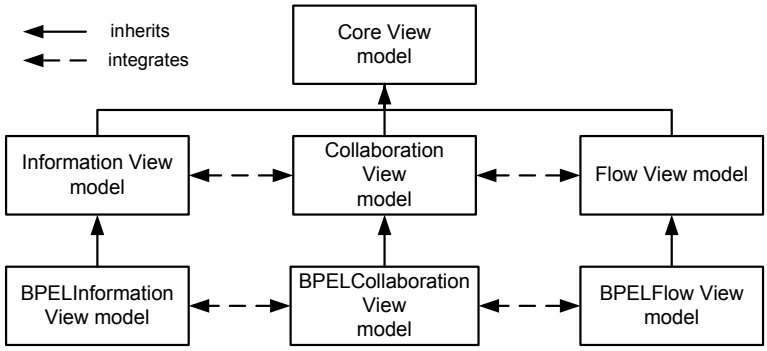


Figure 3.3: View-based Modeling Framework (VbMF)

VbMF consists of modeling elements such as view models, and views. A (view) model (semi-)formally specifies particular business process concerns[132]. A (view) model instance, also referred to as *view* throughout this thesis, conforms to an appropriate (view) model. The models are defined on top of a meta-model. We use the Eclipse Modeling Framework (EMF) meta-model to define our models. Accordingly, as shown in Figure 3.3 the VbMF core model is derived from the EMF[128] *.ecore meta-model[132]. All views depicted in this thesis are based on the XML metadata interchange (XMI) standard[47]. In Figure 3.3, the rectangles depict basic models of VbMF and the lines depict their relationships to each other. Before we describe the relationships between the view models and views, in the following, we overview basic VbMF models.

- The Core View model is the basic VbMF model and is derived from the Ecore meta-model[128].
- The Flow View model describes the control flow of a process.
- The Collaboration View model basically describes the service operations.
- The Information View model specifies the service operations in more detail by defining data types and messages.

Examples for technology-specific models are:

- The BPEL Flow View model is inherited from the abstract Flow View model. This view specifies BPEL specific flow elements such as waits, throws and loops.
- The BPEL Collaboration View model extends the basic Collaboration View and specifies the service operations and channels.
- The BPEL Information View model extends the abstract Information View by concretely specifying BPEL specific elements such as XML Schema (XSD) elements, web service messages, and primitive and complex data types.

A new view model can be *designed*, or *extended* from another view model by adding new features. As displayed by the dashed lines in Figure 3.3, basic VbMF view models, namely the Information View model, the Collaboration View model, and the Flow View model, extend the VbMF Core View model[132]. Moreover, a view can be *integrated* with another view in order to produce a combined view. The dotted lines in Figure 3.3 indicate view integration e.g. the Collaboration View integrates the Information View to produce a combined view. By the mechanism of view integration, views can be enriched by keeping a loose coupling between the views. They can be integrated via view integration points to provide a richer view or a more thorough view of the business process. These view integration points can be determined by a view integration algorithm based on name-based matching[132]. This algorithm integrates entities of one view with matching entities of another view. In the following we shortly describe basic models of VbMF shown in Figure 3.3. Furthermore, VbMF uses *transformations* to generate source code from the views. As VbMF is based on model-driven development (MDD)[139], i.e., platform-specific code, such as BPEL/ WSDL, can be generated from the views[132]. We use the Eclipse model to text (M2T) project's Xpand language[129] to generate source code from the models. A BPEL definition for the process flow and a service description in WSDL[140] are generated from the BPEL Flow View, the BPEL Collaboration View and the BPEL Information View. The Apache Axis2 Web services engine[9] supports us in building Java proxies and skeletons for the services.

To summarize, VbMF focuses on reducing the development complexity in process-driven SOAs. By exploiting the concept of separation of concerns, it enables stakeholders to concentrate on tailored views. By the mechanism of view integration, it is possible to combine these tailored views. In this way, VbMF can, in particular, connect data

types and messages of the BPEL Information View to service operations of the BPEL Collaboration View.

Core-, Flow-, Collaboration-, and Information Flow View model

In the following we specify the basic VbMF models in order to better understand the contributions presented in the following chapters.

Core View model Figure 3.4 illustrates the VbMF Core View model in more detail. The Core View Model defines basic elements such as *Element*, *Identifier*, and *Namespace*. The main *Process* entity contains *1..n Views*, and consists of a list of required and provided *Services*[132].

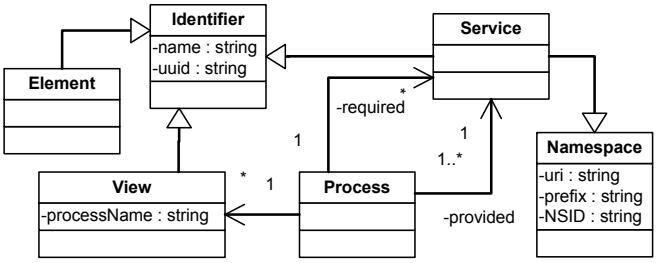


Figure 3.4: Core View Model

Flow View model The abstract Flow View model extends the basic Core View model and specifies a control flow of activities. The BPEL Flow View model is an example of a concrete technology-dependent model extended from the Flow View model.

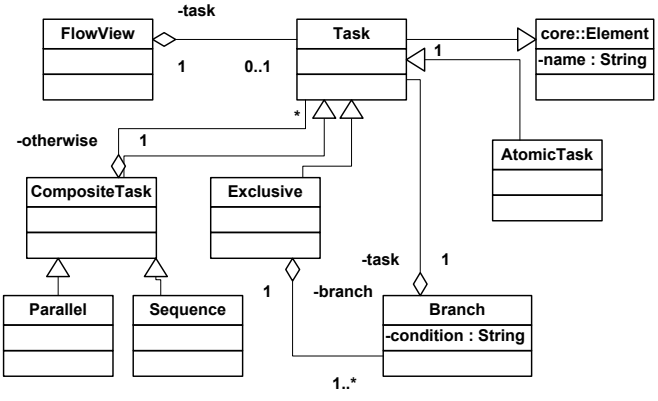


Figure 3.5: Flow View Model

As shown in Figure 3.5, the *FlowView* consists of *0..n* activities of the class *Task*[132]. Please note that in this context we use the words Activity and Task synonymously. There are basically three specializations of the main *Task* class:

1. An entity of class *Exclusive* can contain one or more *Branch* entities defining a condition. Each *Branch* entity in turn can consist of an entity of class *AtomicTask* or *CompositeTask*[132].

2. An atomic task *AtomicTask* models a simple activity within a sequence or a parallel[132].

3. A composite task *CompositeTask* specifies parallels and sequences of activities. A *Sequence* class specifies activities running one after the other whereas the *Parallel* class specifies tasks running in parallel. The containing tasks of parallels or sequences can be of type *AtomicTask*, *CompositeTask* or *Exclusive*[132].

Collaboration View model The abstract Collaboration View model extends the Core View model and basically defines service operations to be invoked by a process activity. Similarly, the BPEL Collaboration View model defines the information needed to generate a web service description language document (WSDL) such as messages.

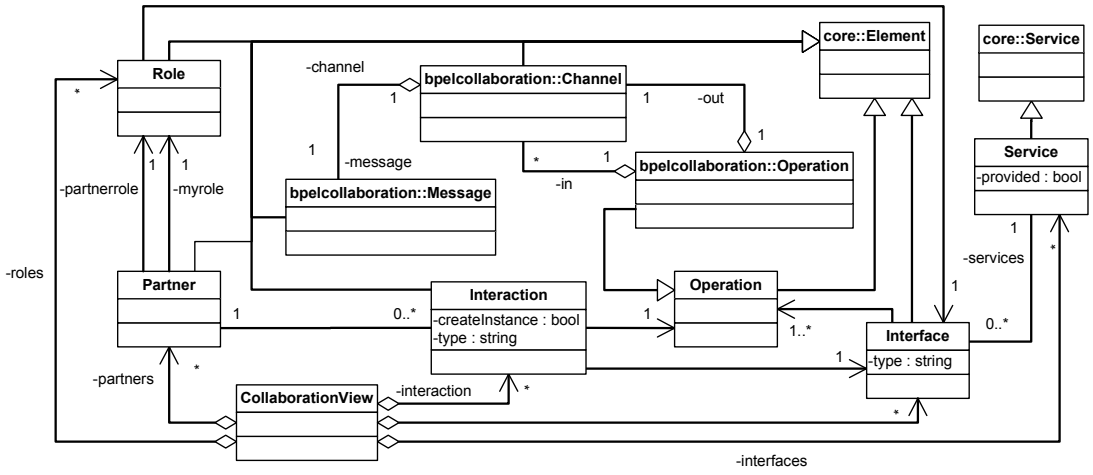


Figure 3.6: Collaboration View Model

In Figure 3.6, we illustrate both the Collaboration View model and an extract of the BPEL Collaboration View model. The *CollaborationView* consists of a list of *Interactions*, *Services*, *Interfaces*, *Roles*, and *Partners* respectively[132].

- An *Interaction* is related to a *Partner*, a service *Interface* and a service *Operation*.
- A *Service* has *0..n Interfaces*.

- A service *Interface* consists of *1..n* service *Operations*.
- A *Role* is related to a service *Interface*.
- A *Partner* has both *0..n Interactions* and its own *Role* and the partner *Role*.

In addition, we describe two entities *Message* and *Channel* of the BPEL Collaboration View in relationship with the basic Collaboration View.

- An *Operation* of the BPEL Collaboration View is extended from the abstract *Operation* of the Collaboration View.
- An *Operation* has an input *Channel* and an output *Channel*.
- A *Channel* is related to a *Message*.

Information View model The abstract Information View model extends the Core View model and specifies the service operations in more detail by defining data types and messages. In distributed systems, data is passed from one system to another. Each system has its own underlying data structures. In particular, the technical BPEL Information View model consists of data types and elements that can be used to generate an XML Schema (XSD).

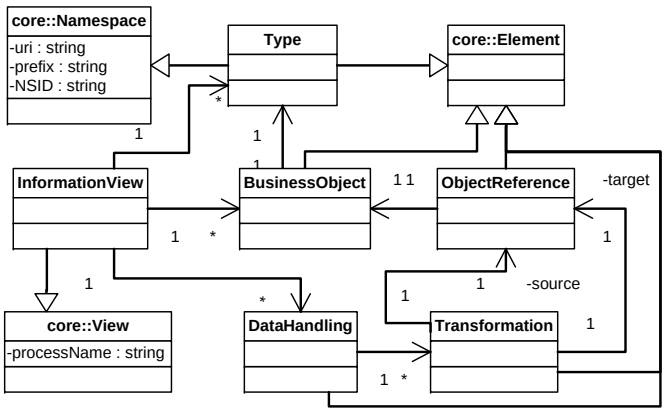


Figure 3.7: Information View Model

The *InformationView* consists of a list of *BusinessObjects*, *Types*, and *DataHandlings* respectively [132] (see Figure 3.7).

- A *BusinessObject* consists of a *Type*.
- A *Type* is a basic element derived from the core elements *Element* and *Namespace*.

- A *DataHandling* consists of *1..n Transformations*. A *Transformation* contains a source and a target *ObjectReference*. An *ObjectReference* simply contains a *BusinessObject*.

CHAPTER 4

View-Based Data Modeling Framework

In this chapter we present our view-based data modeling framework (VbDMF) utilized to implement the concepts presented in the subsequent chapters. This chapter is organized as follows: Firstly, in Section 4.1 we motivate our view-based solution modeling persistent data access. Next, in Section 4.2 we present the underlying basic models and views. In Section 4.3 we describe the models in more detail. Section 4.4 evaluates our models using an industrial case study in the context of a district court. Section 4.5 discusses the contributions and limitations of our approach, and finally, Section 4.6 sums up.

4.1 Motivation

When the number of services as well as the data access services (DAS) in a process-driven SOA grows, the complexity increases along with the number of process elements. Furthermore, when developing and maintaining persistent data access in process-driven SOAs, stakeholders are interested in different concerns of persistent data access. In Figure 4.1, we exemplarily illustrate involved stakeholder roles with their corresponding tasks when modeling persistent data access in business processes. In the following we describe these stakeholder roles in more detail. We reuse these stakeholder roles throughout the remainder of this thesis.

- Business process developer Business process developers need an overview of the fundamental elements of a process flow such as the process activities. As process activities can invoke DAS, business process developers are also interested in basic DAS implementation elements such as the DAS operations, service endpoints, messages, and data types.

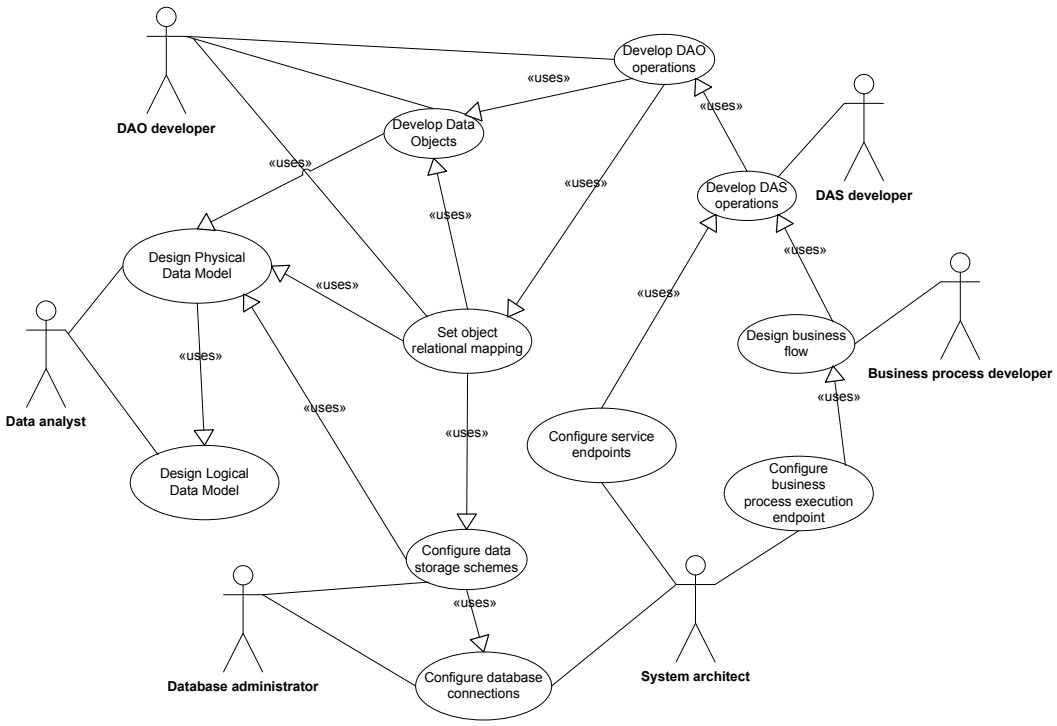


Figure 4.1: Different Stakeholders Focusing on Persistent Data Access in a Process-Driven SOA

- Data access object (DAO) developer DAO developers focus on the DAO operations with the object relational mappings (ORMs) between the database tables and the data objects. For this, they also need an overview of the data storage schemes and database connection properties.

- Data access service (DAS) developer When implementing DAS, the DAS developers need to view descriptions of the underlying DAO operations in order to invoke them. In addition, as the DAO developers, they need to know available database connections e.g. in order to being able to test the DAS. In addition, they need an overview of the DAS and DAS operations.

- Data analyst Data analysts specify logical data models of a real-life use case. In a next step, these abstract logical data models needs to be technically refined by creating physical data models. Moreover, data analysts focus on solving structural problems in business processes concerning data. Hereto, they need an overview of the DAS, DAOs, data objects, object-relational mappings, and database connections within the business process.

- Database administrator Database administrators are responsible for developing

and maintaining the physical data storage schemes and database connections.

- System architect System architects typically focus on the system's runtime configuration, in particular, the business process execution endpoint of the process flow, the service endpoints, and the database connections.

In addition, the following stakeholders solely read-only access to various persistent data access concerns within the business process:

- IT-architect IT-architects focus on the big picture of the business process persistent data access interconnection. On one side, they need to have an overview about the whole architecture, and on the other side, they need to inspect implementation details such as the service provider urls, object-relational mappings, and the database connections.
- Database tester Database testers are responsible for testing persistent data access in a process-driven SOA. They have to create, develop, and run test cases of the persistent data accessing parts of the process. Hence, database testers are in particular interested in the persistent data relevant parts of the business process such as the DAS, and the underlying data storage schemes and the database connections.

4.2 Overview

The view-based data modeling framework (VbDMF) extends the view-based modeling framework (VbMF) (basically described in Section 3.4). Whereas VbMF defines basic processes and services, VbDMF focuses on modeling persistent data access in process-driven SOAs.

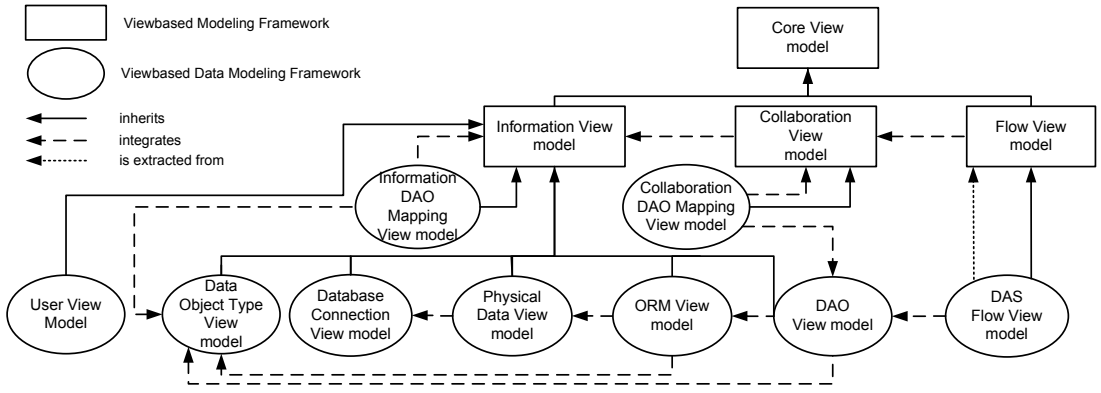


Figure 4.2: VbMF and VbDMF – Overview

Figure 4.2 illustrates the views of VbMF and VbDMF, and their relationships with each other. The rectangles display view models of VbMF, whereas the ellipsoidal boxes denote the additional data-related view models of VbDMF.

VbDMF view models In the following the basic view models of VbDMF are shortly described:

- The Collaboration DAO Mapping View model is an optional view model that maps DAS operations to DAO operations.
- The DAO View model describes the DAO operations, encapsulating persistent data access in object-oriented environments, in detail.
- The DAS Flow View is extracted from the Flow View that only contains the data access activities of the process
- The Database Connection View model comprises a list of arbitrary, user-defined connection properties.
- The Data Object Type View model specifies data object types and data object member variables used to store values in object-oriented environments.
- The Information DAO Mapping View model is an optional view model that maps data types of the services (specified by the Information View) to data object types of the DAOs (specified by the Data Object Type View).
- The ORM View model maps physical data to data object types.
- The Physical Data View model specifies the data storages such as database tables and columns accessed from the DAOs.
- The User View model gives an overview of the registered and published DAS, and the users who registered and published the DAS.

As displayed by the solid lines in Figure 4.2, the new view models of VbDMF represented by the ellipsoids extend basic VbMF views namely the Flow View, the Collaboration View, and the Information View. The dashed lines in Figure 4.2 are used to display view integration, e.g. the Collaboration DAO Mapping View integrates the Collaboration View and the DAO View to produce a combined view.

View extraction Besides *view integration* and *view inheritance*, in Chapter 5, we will introduce a new relationship between views, the mechanism of *view extraction*: A view can be extracted from another view in order to produce a flattened view. The dotted lines in Figure 4.2 display an extraction relationship between the DAS Flow View and the Flow View. The DAS Flow is extracted from the Flow View and, thus only contains the Flow View's persistent data access activities. We go deeper into how to extract a view from another view in Section 5.5.

To summarize, whereas VbMF focuses on reducing the development complexity of business processes and services, VbDMF introduces tailored views for integrating persistent data access into the services of business processes.

4.3 VbDMF Models

In the following, we describe the basic models of VbDMF in more detail. Following VbMF's concepts, we distinguish low-level, technical views from high-level, conceptual i.e. business-oriented views. In addition, our low-level technical view models support separating technology-specific from technology-independent views, both for presenting the information in the models to different stakeholders and for supporting platform-independence via model-driven software development.

Collaboration DAO Mapping View model The Collaboration DAO Mapping View is a technology-dependent view that can be used to map DAS operations of the Collaboration View to DAO operations of the DAO View. Instead of using the Collaboration DAO Mapping View, the Collaboration View could also be integrated with the DAO View by using the VbMF/VbDMF's mechanism of view integration. However, as we use a name-based matching algorithm for view integration, without using the Collaboration DAO Mapping View, the DAO operations and the DAS operations would have to be named identically. Thus, the Collaboration DAO Mapping View model is an optional view.

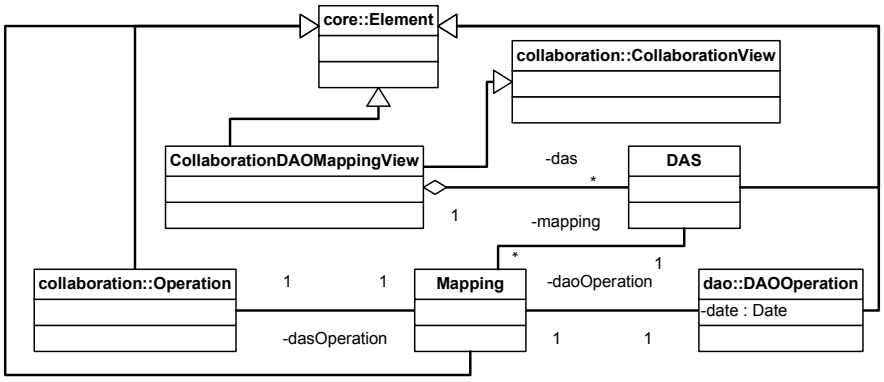


Figure 4.3: Collaboration DAO Mapping View Model

As shown in Figure 4.3, the Collaboration DAO Mapping View model is extended from the Collaboration View model. The Collaboration DAO Mapping View contains a list of *DAS*. Each *DAS* consists of a list of *Mapping* entities. A *Mapping* entity maps an *Operation* of the Collaboration View to a *DAOOperation* of the DAO View.

Data Access Object (DAO) View model The DAO View is a technology-dependent view that basically specifies the DAO operations used to encapsulate access to a data storage. The view can integrate the ORM View and the Data Object Type View and is typically used by the DAO developers.

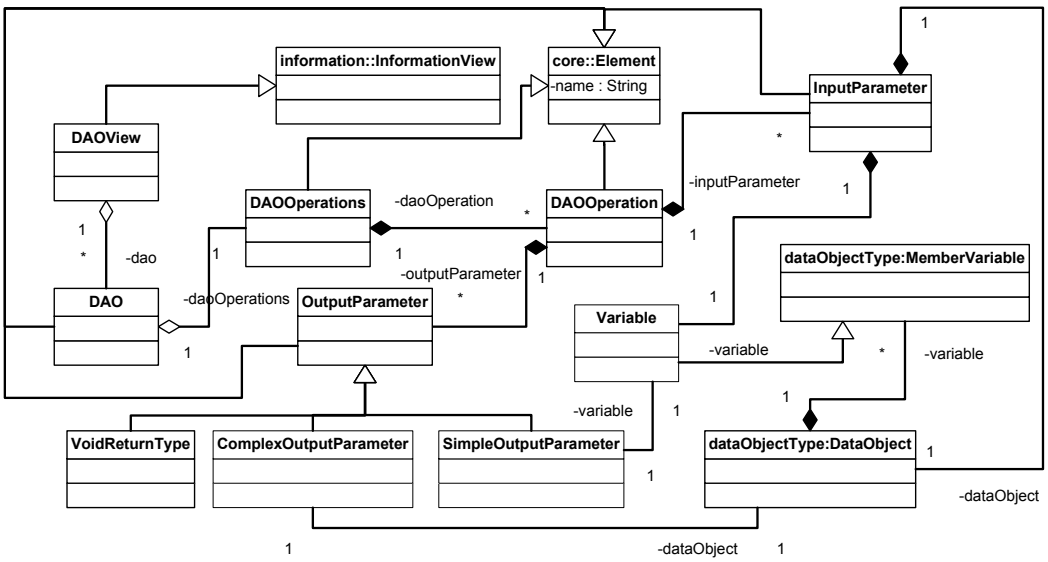


Figure 4.4: Data Access Object View Model

Figure 4.4 illustrates the DAO View that consists of a list of *DAOOperations*. Each *DAOOperation* can consist of a list of *InputParameters* and *OutputParameters*. Each *InputParameter* can consist of a *Variable* or a *Data Object*. An *OutputParameter* can either be a *VoidOutputParameter*, a *SimpleOutputParameter*, or a *ComplexOutputParameter*. A *SimpleOutputParameter* consists of a *Variable* and a *ComplexOutputParameter* contains a *Data Object*.

Please note, that, in this thesis, the DAOs are only one representative for all other types of DAS implementations. As nowadays, the object-oriented programming (OOP) paradigm is typically used to implement services, we use the DAO pattern as exemplary DAS implementation of use throughout this thesis. Moreover, as our concepts are intended for use in larger environments, we propose ORM instead of the more primitive Java Database Connection (JDBC) interface to access the data.

DAS Flow View model As shown in Figure 4.2, the DAS Flow View model is extended from the Flow View model, as it also models a flow. In addition, we can extract the DAS Flow View from the Flow View, because it is the DAS Flow View that only contains the persistent data access activities of the Flow View. In Section 5.5, the DAS Flow View model is described in more detail, and then we further specify how to extract the DAS Flow View from the Flow View.

Database (DB) Connection View model: The Database Connection View comprises a list of arbitrary, user-defined connection properties and therefore is a conceptual rather than a technical view. We also support technology-dependent Database Connec-

tion views through model extension, e.g., a JDBC Database Connection View model for database-independent connectivity[27].

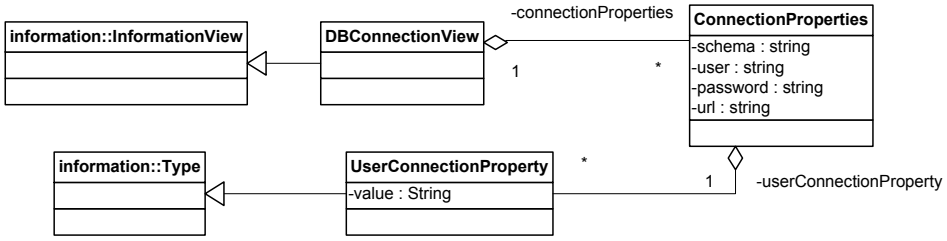


Figure 4.5: Database Connection View Model

As shown in Figure 4.5, the Database Connection View describes the database connections, each comprising a list of *Connection Properties*. Establishing the connection requires at least minimum configuration effort for defining basic properties such as database driver, database url, user, password, etc. The *Connection Properties* contain a list of pre-defined properties and consist of a list of user-defined *UserConnectionProperties*. Typically, they are the system architects, DAO developers, and database administrators who need an overview of the database connections.

Data Object Type View model In object-oriented programming languages, data is defined by ordinary objects[137]. We provide a conceptual, technology-independent Data Object Type View that can be integrated by the ORM View and the DAO View. The Data Object Type View is typically used by the DAO developers. In order to define additional simple data types, developers can extend this view model to gain a technology-dependent (e.g. programming-language-dependent) view.

In object-oriented programming languages information is stored in the objects' member variables. We provide a conceptual, technology-independent model, that consists primarily of a list of *DataObjectTypes*. Each *DataObjectType* in turn consists of a list of *MemberVariables* and/or *DataObjects*. Furthermore, whereas each *DataObject* is of type *DataObjectType*, a *MemberVariable* can be of type *SimpleDataType*.

Flow View model With our new VbDMF Flow View model, besides basic process activities, we can specify a special type of process activities, the persistent data access activities. In Section 5.5 we both describe this view model in detail and explain how to utilize this new VbDMF Flow View in order to solve structural problems in business processes.

Information DAO Mapping View model The Information DAO Mapping View model is an optional technology-dependent view model that maps data types of the services to data object types of the DAOs. In distributed systems, data is passed from

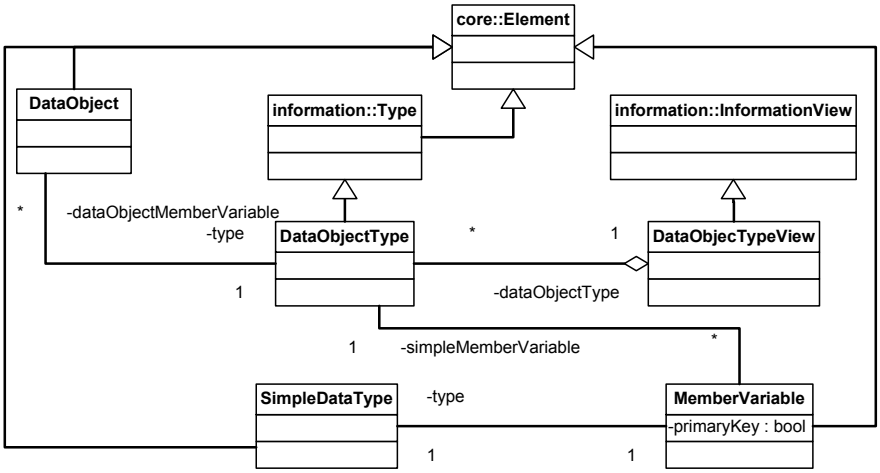


Figure 4.6: Data Object Type View Model

one system to another. Each system has its own underlying data structures. For this purpose we specify data type mappings to support data interoperability between diverse systems: The Information DAO Mapping View specifies conversions between web service description language (WSDL) schema types and data types of the service providers' software system environment. For this purpose, a class *BusinessObjectTypeMapping* associates each *ComplexType* of the BPELInformation View with a *DataObjectType* of the *Data Object Type View*.

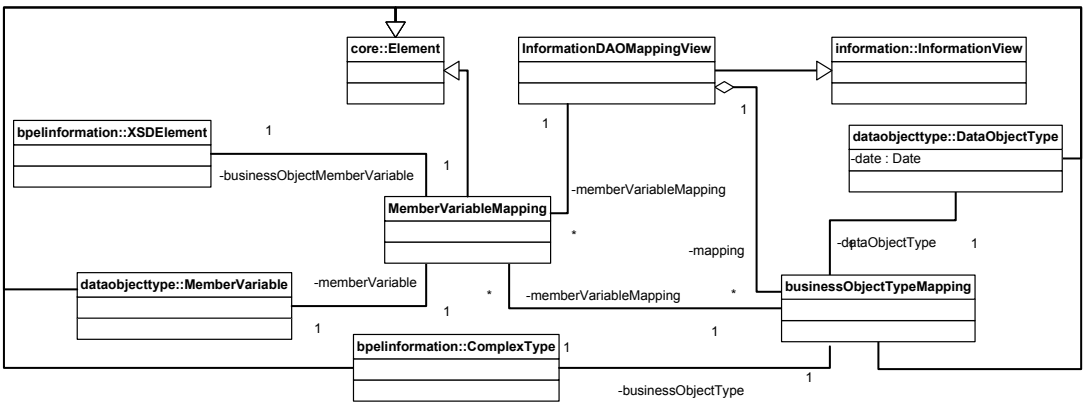


Figure 4.7: Information DAO Mapping View Model

Figure 4.7 illustrates the Information DAO Mapping View model. Each *BusinessObjectTypeMapping* consists of a list of *MemberVariableMappings*. Finally, each *MemberVariableMapping* maps an *XSDElement* of the BPEL Information View to a *Member-*

Variable of the Data Object Type View.

Object Relational Mapping View model The Object Relational Mapping View is a technology-independent model that provides the basis for specifying object relational mapping mechanisms in VbDMF. In this case, technology-independent means, that the ORM View can be described independently from a specific ORM framework such as Ibatis[57] and Hibernate[55]. In order to specify certain features of specific ORM frameworks, DAO developers should design a new technology-dependent model by model extension.

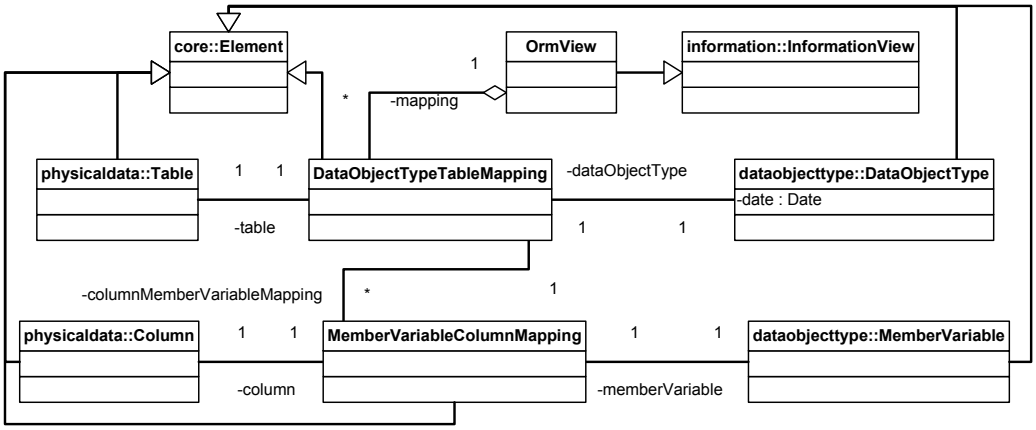


Figure 4.8: ORM View Model

The basic ORM View describes mappings between data object types and member variables of the Data Object Type View and database tables and columns of the Physical Data View. Accordingly, the class *DataObjectToTableMapping* maps a data object type (*DataObjectType*) to a database table (*Table*). The class *MemberVariableToColumnMapping* allows for a more specific mapping between a *MemberVariable* and a table *Column*.

Physical Data View model This view model is primarily intended for data analysts, DAO developers, and database administrators who rely on detailed physical database design. As shown in Figure 4.2, the Physical Data View can integrate the Database Connection View.

The Physical Data View contains two basic classes: *Tables* and *ColumnTypes*. In particular, the class *Table* comprises a list of *Column* and *RefColumn* types. We use the class *RefColumn* to model referential constraints between tables. We support most common simple data types for current RDBMSs. As data types can differ among different RDBMSs, developers can create a new technology-dependent Physical Data View by extending this conceptual view model.

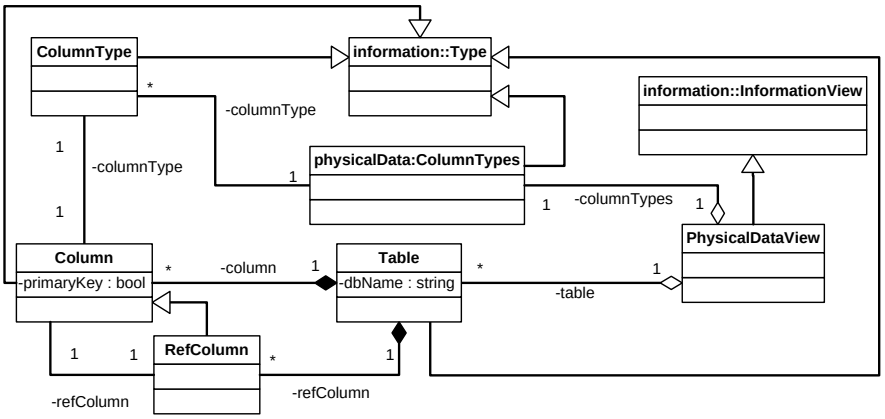


Figure 4.9: Physical Data View Model

User View model The User View is extended from the Information View and gives an overview of the users who register and publish DAS to service repositories. In Section 7.3, we describe the User View in more detail. In addition, in Section 7.2, we illustrate how stakeholders can register and publish DAS using our view-based model-driven data access architecture (VMDA).

4.4 Case Study

To illustrate the applicability of VbDMF, in this section, we present a case study which we will refer to again in Chapter 5.

This case study deals with a real workflow of a specific e-government application modeling the jurisdictional provisions in the context of a district court. However, the applicability of our VbDMF is not limited to this type of applications. VbDMF can reasonably be applied to all applications, based on a process-driven SOA, where data is accessed from a persistent storage.

First of all, let us explain the business process flow at the land registry court illustrated in Figure 4.10. As governmental processes are typically very complex[107], for reasons of simplicity, we use a flattened workflow for demonstration. We use a UML[101] activity diagram to model the process flow. Each process activity contains basic actions, the fundamental behavior units of an activity[101]. The business process consists of different types of actions, namely service operations, data access service operations, transformations, and human actions. The process starts when a new jurisdictional application is received. Then, the *ValidateApplication* activity invokes a service that checks the incoming jurisdictional application for correct syntax and semantic. Successfully validated applications are saved by a flow of alternate transformation activities and persistent data access activities. In case the validation fails, neither data is stored nor the delivery is sent to the applicant. In order to store data into the database by object rela-

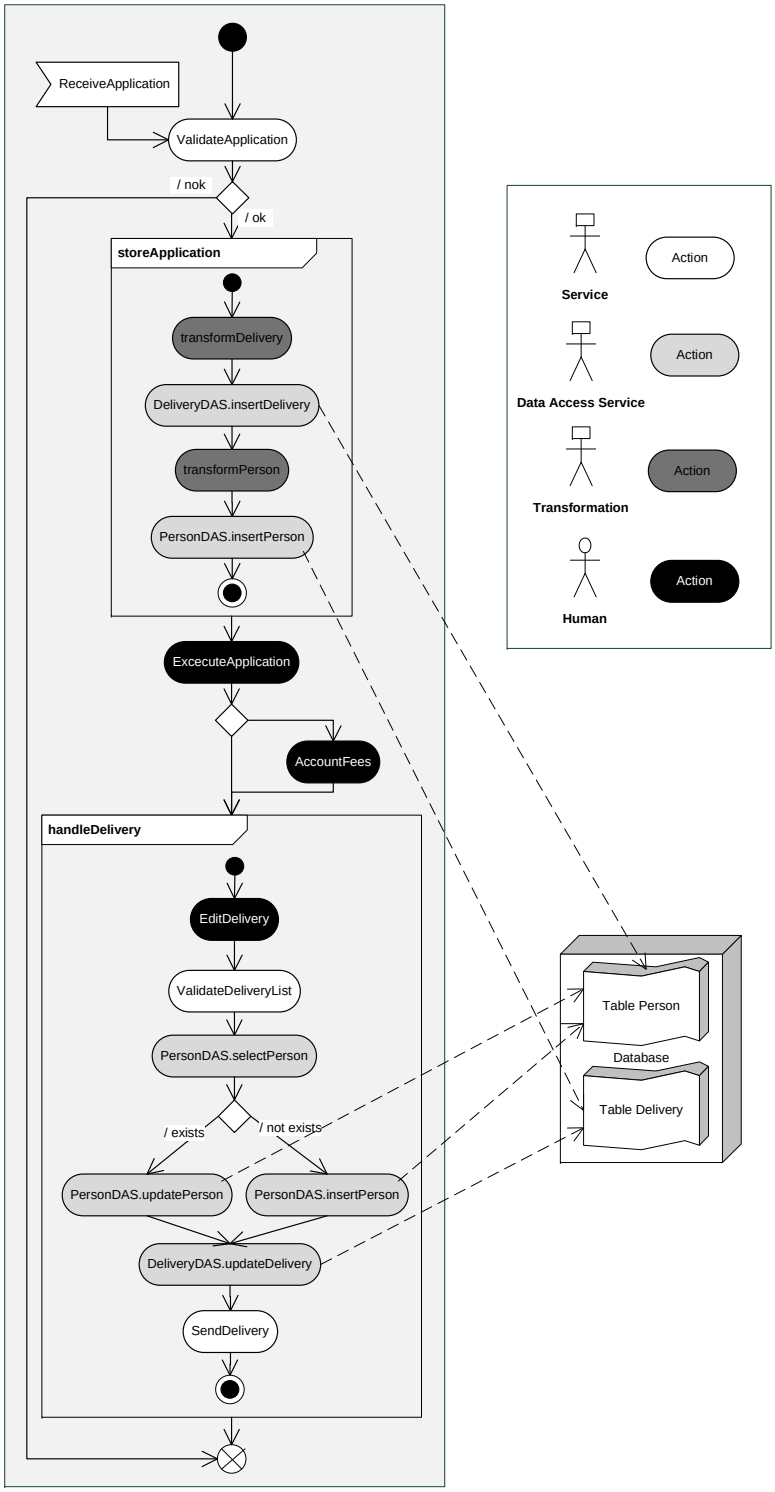


Figure 4.10: Case Study: Simplified Process Flow at the Land Registry Court

tional mapping (ORM) mechanisms, the process data need to be transformed into data objects. The activities *transformDelivery* and *transformPerson* transform delivery and applicants process data respectively into associated data objects. After executing each of these transformation activities, the persistent data access activities *insertDelivery* and *insertPerson* respectively are invoked in order to persistently store the resulting data objects. Stored applications can be executed by the registrar within the human process activity *ExecuteApplication*. If the registrar approves the application, the service-based activity *AccountFees* will be invoked. As a dismissed application is free of charge, the service operation *AccountFees* is never invoked in case of dismissal. After accounting the fees, the registrar has to select whether the approval or dismissal shall be delivered by the system. Dependent from the registrar's decision, the approval or dismissal is delivered to the applicant. For this purpose, the process activity *ValidateDelivery* checks the recipient information for correctness and completeness before sending the delivery to the applicant. In case of successful validation, the two DAS operations *updatePerson* and *updateDelivery* are invoked in order to store the recipient information persistently. If the validation fails, the persistent data access activity *selectPerson* will return zero rows. In this case, instead of updating the person, a new person has to be inserted by invoking the persistent data access activity *insertPerson*. Finally, the service operation *SendDelivery* sends the delivery to the recipient by invoking an external service.

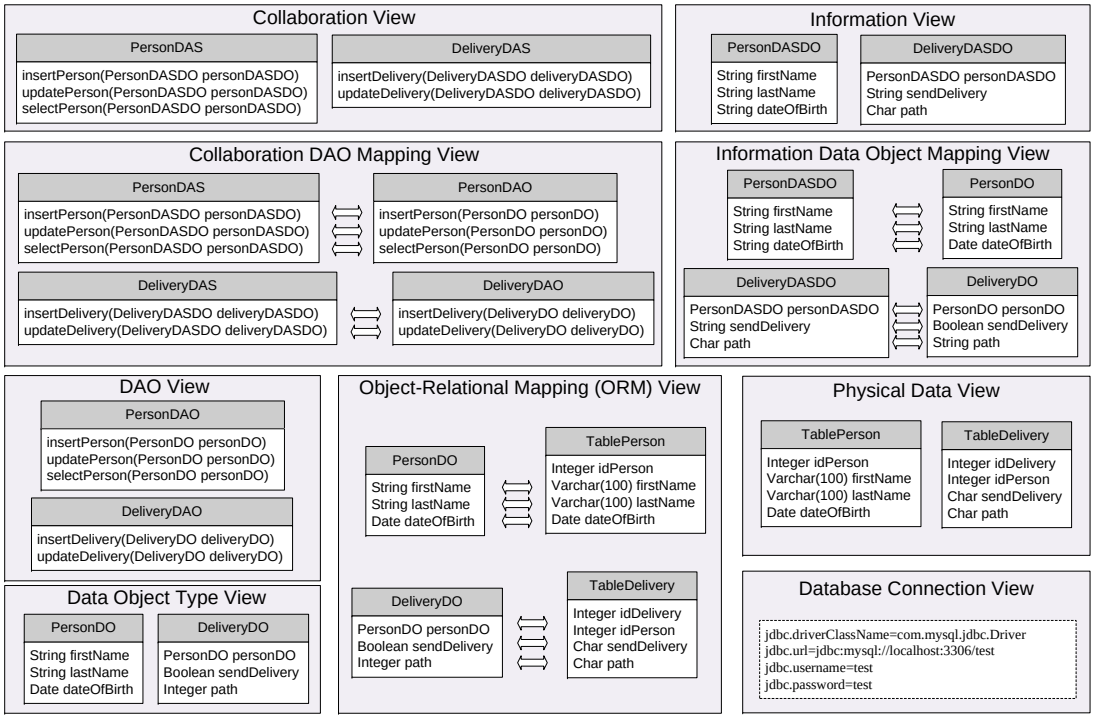


Figure 4.11: Case Study: Illustration of the VbDMF views

In the following we apply our case study to illustrate and verify our VbDMF models.

VbDMF views illustration Firstly, we illustrate the VbDMF models from the stakeholders' rather than from the model developers' point of view. Figure 4.10 displays a Flow View with various persistent data access activities namely *DeliveryDAS.insertDelivery*, *PersonDAS.insertPerson*, *PersonDAS.selectPerson*, *PersonDAS.updatePerson*, and *DeliveryDAS.updateDelivery*. In Figure 4.11, we denote the relationships between the Flow View and the data-related views of its persistent data access activities. As depicted in Figure 4.11, the Collaboration View basically defines the DAS operations invoked by the process activities of the Flow View. According to the WSDL [140] specification, each DAS operation consists of an input and/or output DAS message. These DAS messages together with the data object types (*PersonDASDO* and *DeliveryDASDO*) are in turn defined by the Information View. The DAS operations above are mapped to the DAO operations *PersonDAO.insertPerson*, *PersonDAO.updatePerson*, *PersonDAO.selectPerson*, *DeliveryDAO.insertDelivery*, and *DeliveryDAO.updateDelivery*, respectively, by using the Collaboration DAO Mapping View. These DAO operations in turn are described in detail by the DAO View. Each DAO operation references some data object types (*PersonDO*, *DeliveryDO*) used to encapsulate data in object-oriented environments. A Data Object Type View specifies these DAO data object types. The DAO data object types can be mapped to the data access service's data object types by using the Information DAO Mapping View. In addition, each of the DAO data object types (*PersonDO*, *DeliveryDO*) is mapped to a specific database table (*TablePerson*, *TableDelivery*) by a object-relational mapping (ORM) mechanism, as shown in the object relational mapping View. There are a large number of both commercial and open-source tools for mapping data object types to database tables. Two common example of open-source tools are Hibernate [55] and Ibatis [57]. This object-relational mapping is shown by the ORM View. In order to illustrate which DAO operation accesses which database tables, data analysts and DAO developers can integrate the ORM View into the DAO View. Each table of the ORM View can be integrated with the database table definitions of the Physical Data View. Database test developers are probably interested in the Database Connection View in order to check the current database configuration before creating certain test cases.

VbDMF views in XMI notation In the following figures we outline the VbDMF views in XMI notation. As mentioned in Section 3.4, views can be integrated via view integration points. These VbDMF view integration points are displayed as red frames and marks, respectively. In addition, we use a name-based matching algorithm for view integration. Accordingly, a red-framed element in one view acts as view integration point to a red-framed, equally named view integration point of another view. In a uniform manner, the red-labeled elements in two different views can act as view integration points between two views.

As shown in Figure 4.12, we can integrate each *AtomicTask* of the Flow View with further service operation definitions from the Collaboration View.

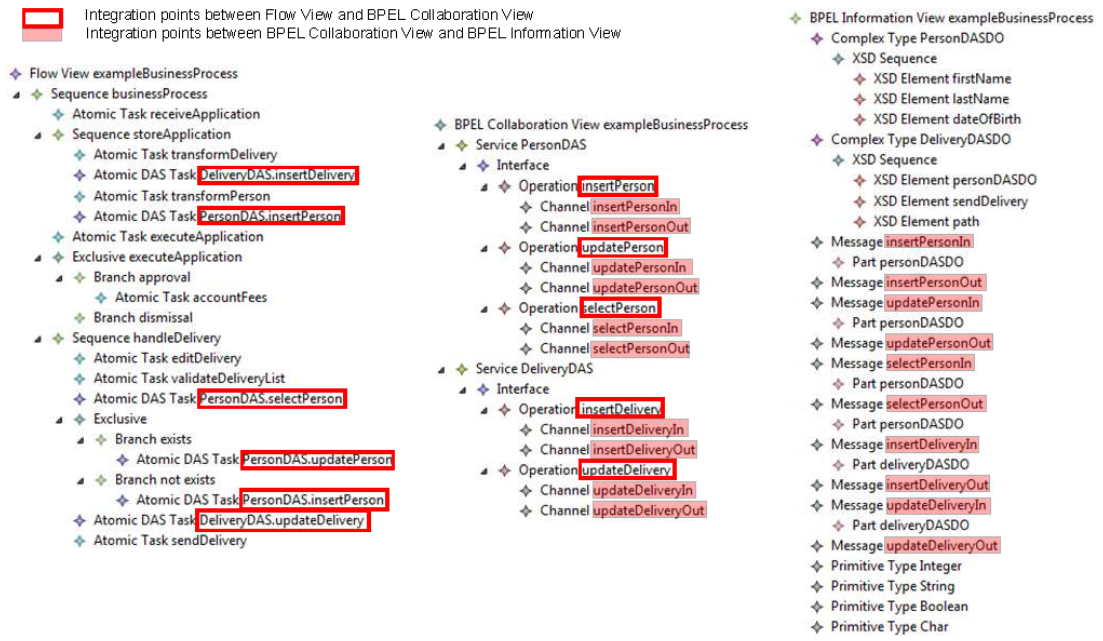


Figure 4.12: Case Study: Flow View, Collaboration View, and Information View in XMI Notation

In particular, each *AtomicDASTask* of the Flow View can be integrated with a DAS operation definition specified by the Collaboration View. Whereas the Collaboration View specifies the service operation definitions of the DAS operations, and the Information View models the underlying messages and data types.

Figure 4.13 illustrates the relationships between the Collaboration View, the Collaboration DAO Mapping View, and the DAO View. The DAO View models the underlying DAO operations of the DAS messages specified by the Collaboration View.

In order to map *AtomicDASTasks* of the Flow View to their underlying DAO operations of the DAO View, we use the Collaboration DAO Mapping View. To exemplify this, we show an XMI view of both the DAO View and of the Collaboration DAO Mapping View in Figure 4.13. Each DAO operation in the Collaboration DAO Mapping View matches a corresponding DAO operation in the DAO View. Accordingly, each DAS operation of the Flow View corresponds to a DAS operation in the Collaboration DAO Mapping View. The Collaboration DAO Mapping View maps DAS operations to DAO operations e.g. the DAS operations *DeliveryDAS.insertDelivery* and *PersonDAS.insertDelivery* of the DAS Flow View are mapped to the DAO operations *DeliveryDAO.insert* and *PersonDAO.insert* of the DAO View.

In Figure 4.14 we depict the mapping between the Information View and the Data Object Type View. We use the Information DAO Mapping View to relate business objects of the service environment to data object types of the object-oriented environment.

Figure 4.13: Case Study: Collaboration View, Collaboration DAO Mapping View, and DAO View in XMI Notation

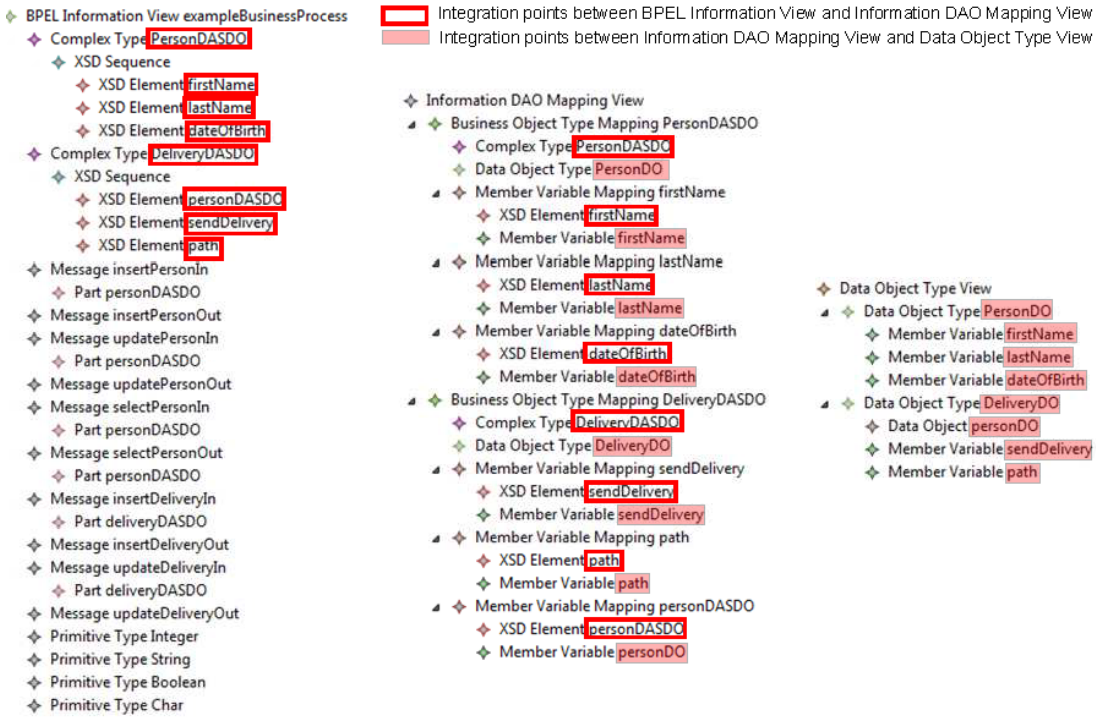


Figure 4.14: Case Study: Information View, Information DAO Mapping View, and Data Object Type View in XMI Notation

As displayed in Figure 4.15, the DAO View can integrate the Data Object Type View in order to get the detailed specification of the data object types.

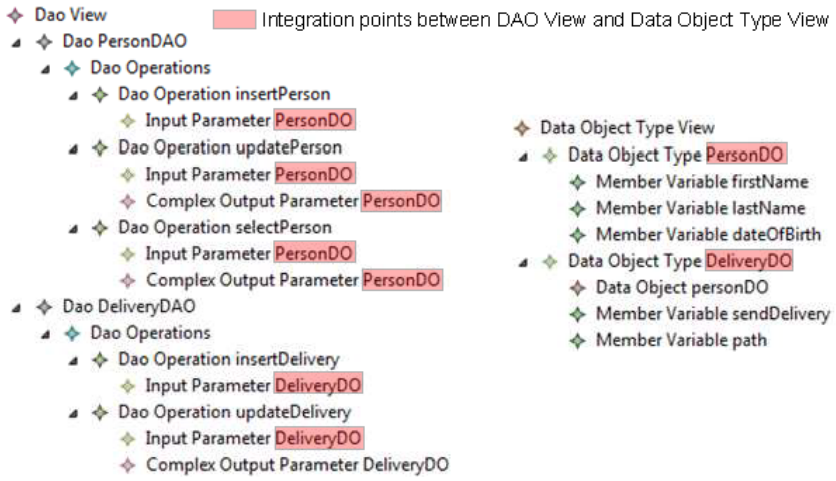


Figure 4.15: Case Study: DAO View and Data Object Type View in XMI Notation

Each *DAO Operation* of the DAO View contains *DAO Input Parameters* and *DAO Output Parameters*. In addition, each parameter type can be mapped to corresponding *Data Object Types* of the Data Object Type View. Please note that, in the figures, the Data Object Type View only contains the name of each parameter rather than the data parameter type. Thus, the data parameter type is specified as attribute of the entities *DAO Input Parameter* and *DAO Output Parameter*, respectively.

The ORM View, displayed in Figure 4.16, integrates the Physical Data View with the Data Object Type View. Accordingly, the ORM View maps *Tables* to *Data Object Types* and *Table Columns* to *Data Object Member Variables*.

The Physical Data View integrates the Database Connection View (Figure 4.16) in order to get the *DBConnectionProperties* details of the underlying RDBMS such as url, user, password etc. The integration of the Physical Data View with the Database Connection View is possible by the entity *Table* of the Physical Data View incorporating a *dbName* attribute acting as view integration point.

4.5 Discussion

In the following we discuss the contributions and limitations of VbDMF used to model persistent data access in process-driven SOAs.

VbDMF is a framework consisting of various views modeling persistent data access tailored to the requirements of different stakeholders. As a result, according to the pattern of separation of concerns, different types of stakeholders can concentrate on their concerns of persistent data access. Because of the model-driven approach, VbDMF

Figure 4.16: Case Study: Data Object Type View, ORM View, Physical Data View, and Database Connection View in XMI Notation

aims at improving software quality and reuse of persistent data access in process-driven SOAs[139].

Our prototype framework uses DAS/ DAOs as example implementation. It specifies persistent data access services (DAS) with underlying data access objects (DAO). Each DAO is based on object-relational mappings (ORM) of data objects to physical database tables. A limitation of our protoype framework is, that the ORM View solely supports *1..1* mappings between one data object and one table. More complex mappings are beyond the scope of this thesis. Accordingly, *1..n* and *n..n* mappings have to be modeled by a list of *1..1* mappings.

With our model-driven approach, we can hide technical details such as ORM-dependent (HIBERNATE[55], IBATIS[57]) configurations such as performance tuning settings from the view modelers. Thus, with VbDMF, we can enhance the level of abstraction for developing persistent data access. It goes without saying that VbDMF is not limited to DAS and DAOs, it can be flexibly extended with new conceptual and technical models.

However, it is a big challenge to balance the flexibility of traditional software development with the high abstraction level of MDD. The more attributes are specified additionally, the more flexible, but also the less abstract are the view models. With our VbDMF, we both support conceptual abstract views (e.g. the Collaboration View) and technology-dependent views (e.g. the BPEL Collaboraton View). The technology-dependent BPEL Collaboration View is extended from the abstract Collaboration View in order to specify BPEL-specific elements such as messages and channels.

We have built a uniform framework for structurally specifying persistent data access in process-driven SOAs. By using VbDMF, DAS can be developed by using homogeneous

programming languages, technologies, and tools. Hereby, we can reduce the development complexity, and thus, increase development productivity and software quality. Moreover, DAS are relatively small, self-contained software components specified to read and write data from a persistent storage. Because of these characteristics, DAS are especially suitable for model-driven source code generation. Moreover, in VbDMF, we do not need to specify each view of VbDMF in order to generate the persistent data access source code. For instance, the DAO View is an optional view, because the DAOs can also be automatically generated from the Collaboration View, the Information View, the Information DAO View, and the Data Object Type View. However, by modeling the DAO View, we can specify the exact names of the DAO operations.

Besides automatic source code generation, VbDMF provides a structured integrated documentation from the services to the physical database storages. Because of that, we have documented which process activities read or write from which database tables, or which DAS run on a certain service endpoint.

4.6 Summary

In this chapter we presented our view-based data modeling framework (VbDMF) as an extension of the view-based modeling framework (VbMF). VbDMF consists of a set view models specifying persistent data access in process-driven SOAs. As our view models are based on VbMF, each model represents a specific tailored view to the requirements of different stakeholders. We evaluated our models by an industrial case study in the fields of e-government. The VbDMF view models as well as the case study will be used to illustrate and verify our concepts in the following chapters.

CHAPTER 5

Improving Traceability of Persistent Data Flows in Process-Driven SOAs

In this chapter we present a concept to improve traceability of persistent data access in process-driven SOAs which can be applied to solve structural problems such as deadlocks in data-intensive business processes. With our view-based model-driven approach, we provide a solution to generate flows of persistent data access activities (which we refer to as persistent data access flows). To the best of our knowledge these persistent data access flows are not used to solve structural problems in process-driven SOAs, yet. Moreover, our persistent data access flows can be flattened by diverse filter criteria e.g. by filtering all activities reading or writing from a specific database or table. In a series of motivating scenarios we show how our persistent data access flow concept can contribute to enhance documentation, traceability, and productivity in service-oriented, process-driven environments.

This chapter is organized as follows: First, in Section 5.1 we shortly motivate our concepts. Next, Section 5.2 provides some background information to better understand the contributions of our approach. In Section 5.3 we give a basic overview of our approach. Next, in Section 5.4 we illustrate how our persistent data access flow concept can solve structural problems in business processes by presenting selected use cases. Section 5.5 describes the details necessary to realize our approach within a model-driven environment: the model-driven specification, integration, and extraction of DAS Flow Views. Section 5.6 demonstrates the applicability of our model-driven solution and presents a suitable tooling, and in Section 5.7, we evaluate the correctness and complexity of the presented algorithms. Afterwards, we discuss the limitations of our approach in Section 5.8. Finally, Section 5.9 summarizes and concludes.

5.1 Motivation

A common problem in business process modeling is the detection of structural errors [119]. Current business process modeling systems (BPMS) [142] lack support for verification of structural problems concerning persistent data access. In many BPMS, such as IBM Websphere MQ Workflow, the process activities cannot request persistent data directly [118]. Therefore, these systems cannot trace persistent data access without the help of external dependencies. In other BPMS, such as Webmethods [123], the process activities are able to invoke persistent data accesses directly. However, they lack tool support for solving structural persistent data access problems at modeling time.

While collaborating on several service-oriented software development projects in a large enterprise, we identified a series of structural problems in business processes concerning persistent data access. All these problems have in common that data is accessed from persistent storage. Three groups of stakeholders are particularly faced with these problems: the data analysts, DAS developers, and the database testers. In the following we shortly describe the drawbacks from the perspective of each of these stakeholders when analyzing, developing, and testing process flows.

Data analysts dealing with deadlock prevention The first group of stakeholders, the data analysts, has to deal with various analysis problems. As an example of one of the many tasks data analysts have to deal with, let us consider a process instance failing at runtime due to a deadlock. This deadlock is caused by a structural problem concerning modeling persistent data access in a business process. Common BPMS can solely trace the specific process activities causing the failure. However, they do not support detection of underlying business process modeling errors. In order to solve the structural problem, data analysts usually have to examine the DAS operations of a process flow by manually stepping through each activity. Therefore, if the flow consists of a large number of activities, this analysis will be rather exhaustive [88].

DAS developers modeling persistent data access activities A second group of stakeholders, the DAS developers, focuses on modeling persistent data access activities. DAS developers also need to add, adapt or delete persistent data access activities of the process flow. For this, DAS developers need to overview the flow of data access activities of the process. However, the persistent data access activities are tightly coupled with other activities. Thus, when the number of control-flow constructs such as switches, loops, and joins [21] as well as the number of process activities grows, to manually overview the required persistent data access in the whole business process can be a complex task. In addition, personal factors as well as the amount of theoretical modeling knowledge influence the ability to understand process models [88].

Database testers creating, developing, and running test cases The database testers are a group of technical stakeholders focused on an important part of the entire business process: To ensure the correctness of the persistent data access activities of

a process flow. In order to create, develop, and run test cases, the test developers have to acquire knowledge about the persistent-data-accessing parts of the technical sub-processes. Even more, the persistent data access activities should be separately testable. Thus, stakeholders need an overview of the control flow between the persistent data access activities. However, up-to-now current BPMS provide no suitable tooling documenting the control flows of persistent data access activities in business processes.

In Section 5.4, we will present conceptual solutions solving each these selected problems.

5.2 Background

Data Flow vs. Control Flow

Common graphical process modeling languages and business process management systems (BPMS)[142] can differentiate between the control flow and the data flow of a process. Examples of graphical modeling languages are the Business Process Model and Notation (BPMN)[100] and the Unified Modeling Langage (UML)[101] activity diagrams. An example of a BPMS is the IBM Websphere MQ Workflow[59] Whereas the control flow describes the sequence of activities of the process flow, the data flow describes incoming and outcoming data to and from process activities. An example of a control flow is depicted in Figure 4.10.

In BPMN, data is transferred in data objects that can be associated with activities. The data flow is modeled by associations from data objects to activities or vice versa. Accordingly, data objects written by one activity can be read be the subsequent activity. In IBM Websphere MQ Workflow, a data flow is modeled by connecting the activity's input and output container. Special data flow connectors define the mapping of the activity's input and output container. In UML 2.0, the data flow is specified by pin elements representing the inputs and outputs of activities. Whereas input pins provide the activities with data, output pins get the data from the activities. Figure 5.1 depicts a data flow in UML notation of the business process in case study Section 4.4.

Lang defines that data flows between processes may represent either attributes of objects, transient data or persistent data[75]. In contrast to these data flows, in this thesis, we concentrate on persistent data access flows. Our persistent data access flows are control flows that solely consist of data access activities reading or writing from a persistent data storage. In contrast, whenever we refer to data flows, we outline the common data flows, representing transient and persistent data respectively, as defined by Lang[75].

Microflow and Macroflow Pattern

Our work is in particular based on the so-called Macro-Microflow pattern[53,54]. The Macro-Microflow pattern is a pattern designed for process-oriented integration in service oriented architectures. According to this Macro-Microflow pattern, a microflow

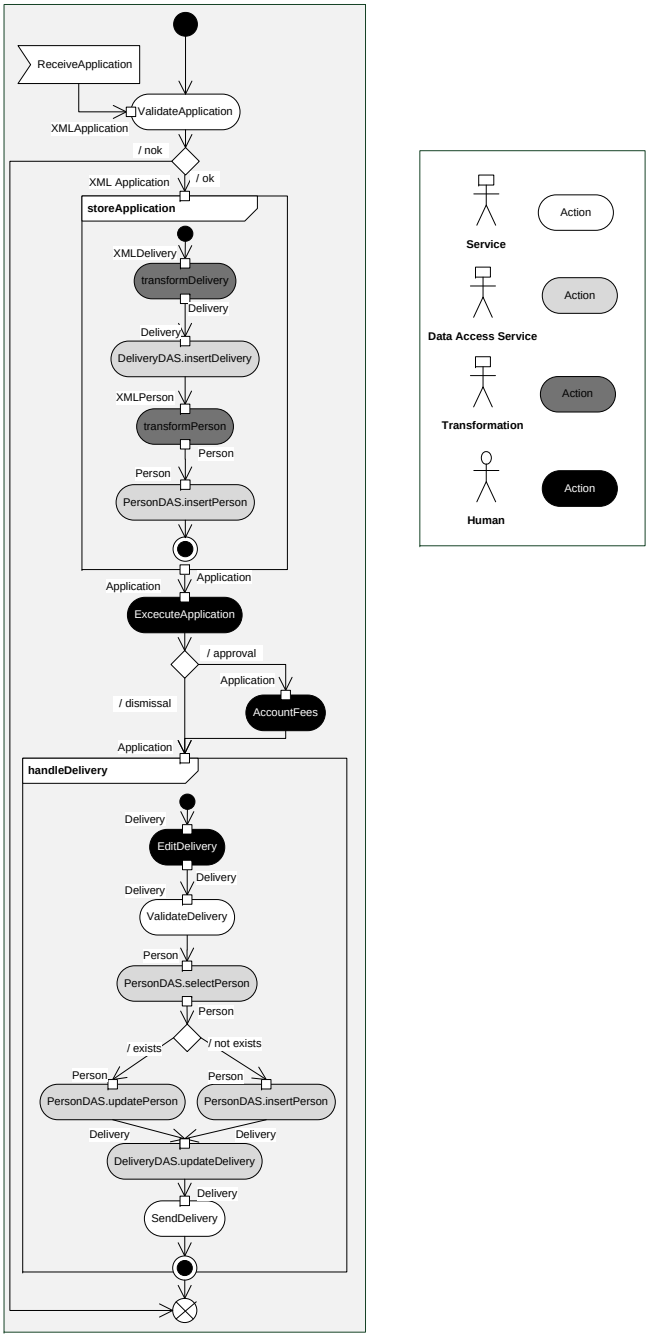


Figure 5.1: Data Flow of a Business Process specified with UML Pin Elements

represents a sub-process that runs within a macroflow activity [53,54]. Macroflows are considered to be high-level conceptual business processes whereas microflows are technical information processes [44,53,54]

There are two types of microflows. Firstly, a short-running technical process that runs automatically and secondly, a flow of activities that can contain interrupting process activities such as human tasks and events. The first alternative, the technical microflows are not interruptible and are running in a transaction [53,54]. The interruptible microflows in turn can contain automatically short-running microflows. When analyzing, developing, and maintaining persistent data access, stakeholders have to focus on these microflows. In Figure 5.2, we will depict two technical microflows as technical sub processes of the whole business macroflow depicted in Figure 4.10 in the case study Section 4.4.

5.3 Overview

In this section we present the basic idea of our persistent data access flow concept. For this, we reuse the business process presented in the precedent case study Section 4.4.

On the left and on the right of Figure 5.2, the resulting persistent data access flows from the business process in the middle are shown. We define persistent data access flows as control flows containing the persistent data access activities of the whole business process flow. We differentiate simple persistent data access flows from filtered persistent data access flows.

- Simple persistent data access flows are control flows containing all and only the persistent data access activities of a business process

- Filtered persistent data access flows are control flows containing only those persistent data access activities of a business process that match certain persistent data access filter criteria

On the left of Figure 5.2, a simple persistent data access flow is depicted. On the right of the figure, a filtered persistent data access flow is shown. The filtered persistent data access flow in this example contains only those persistent data access activities reading or writing data from table *Person*.

In this thesis we use DAS with underlying DAOs as example implementation. However, our approach can be easily applied for other types of persistent data access implementations. In the following we show how our persistent data access flows depicted on the left and on the right of Figure 5.2 can be applied to enhance traceability and documentation of persistent data access in process-driven SOAs. For this purpose, in the following Section 5.4, we present selected problems and solutions from different stakeholders' point of view, in particular from the perspective of data analysts, DAS developers, and database testers.

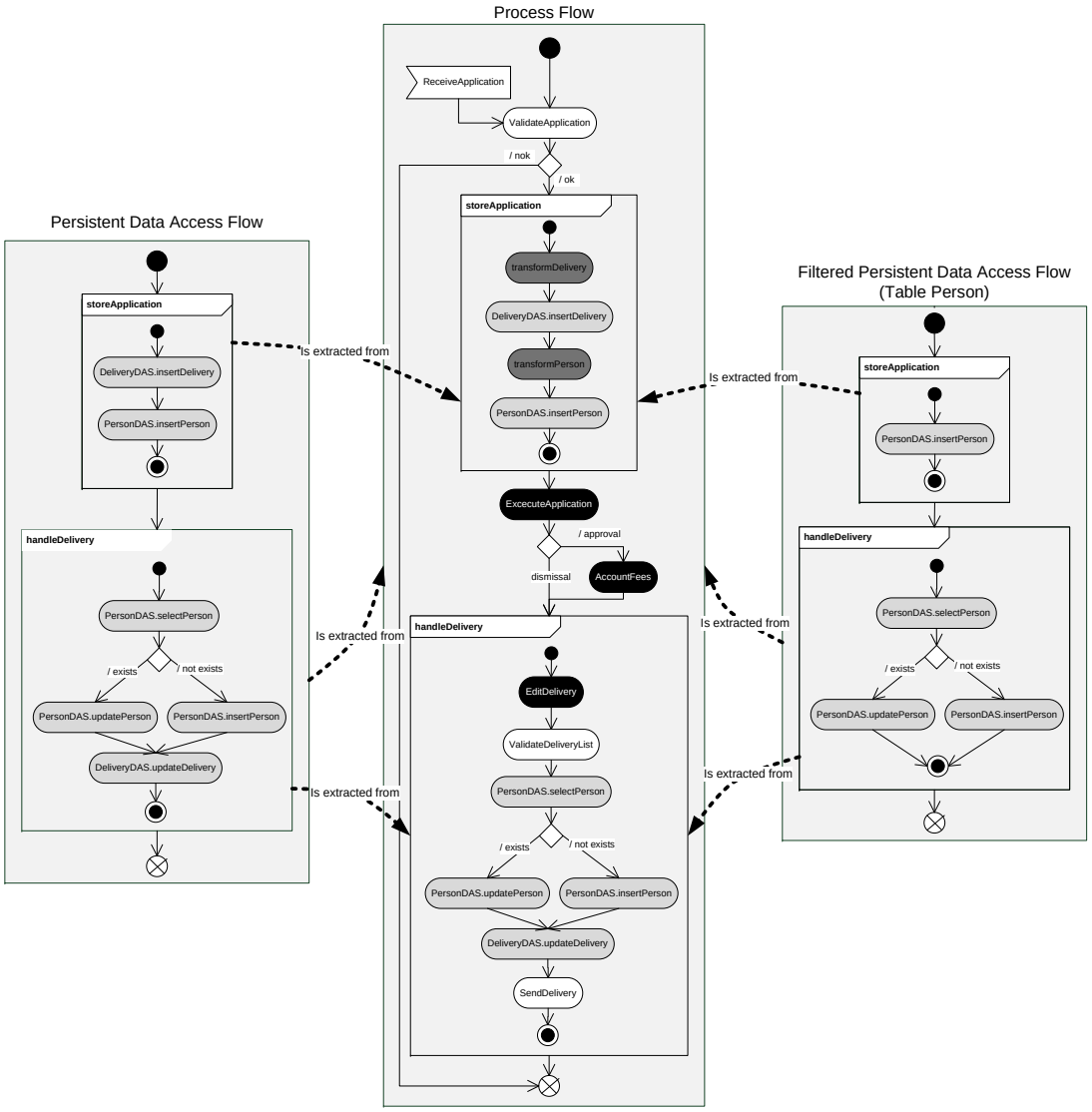


Figure 5.2: Two Persistent Data Access Flows Extracted from a Business Process Flow

5.4 Solving Structural Problems in Business Processes

In this section we illustrate how our persistent data access flow concept can be generalized to solve various data analysis problems. For this, we refer to the selected problems introduced in the motivating Section 5.1. These problems reoccur in many cases for data analysts, DAS developers, and database testers when analyzing, developing, and testing persistent data access in business processes. For each selected problem, we describe how stakeholders can apply our persistent data access flow concept to solve it.

1. At first, we have a look at a typical data analysis problem. We show how our persistent data access flow concept can ease the manual and automated data analysis in process-driven SOAs. Our goal is not to reinvent deadlock detection, but instead show how both *manual and automatic deadlock detection* in a complex process model can be eased by applying the persistent data access flows. On top of our approach existing data analysis solutions such as deadlock detection techniques can be applied.

2. Secondly, we show how DAS developers can benefit from our view-based approach. The persistent data access flows can be applied to document the persistent data access flows in a process. Furthermore, we illustrate how to *detect design weaknesses* concerning persistent data access at the earliest possible state of the development process[28] – in the modeling phase.

3. Thirdly, we describe how our approach provides database testers with appropriate input/output data needed for *test case generation and execution*. Moreover, we explain how the persistent data access flow concept can improve the database testers' documentation. Finally, we illustrate how our persistent data access flows support testers in locating errors more quickly.

Problem & Solution: Deadlock Detection

In process-driven SOAs usually a large number of process instances run in parallel in a process-engine. These process instances often require access to competing data resources such as data from an RDBMS. Deadlocks arise when process instances hold resources required from each other. When none of these process instances will lose control over its resources, a classic deadlock situation occurs[61]. There are various deadlocks detection techniques in order to discover and resolve deadlocks. One common method to resolve deadlocks are database transaction timeouts as used by common database drivers such as the Java Database Connectivity (JDBC) driver[27]. Accordingly, after the timeout expired, process instances lose control over the held resources.

A process can perform some transformations, invoke service operations, and access the database. In order to prevent, detect, and solve deadlocks, data analysts need to focus on the persistent data access activities of a process. Moreover, stakeholders have to make sure that the DAS operations of different process flow instances always have

to be processed in the same order such that no two DAS operations have to wait for competing resources.

Manual deadlock detection with persistent data access flows In the following we present how our approach can contribute to detect deadlocks in business processes by using our persistent data access flow approach. Figure 5.3 displays the two persistent data access flows of our business process. In order to identify the persistent data access activities they are consecutively numbered.

The persistent data access flow on the left hand side simply consists of two DAS operations. The first DAS operation *DeliveryDAS.insertDelivery* (1) inserts delivery data into table *Delivery*. Afterwards the DAS operation *PersonDAS.insertPerson* (2) inserts person data into table *Person*. The persistent data access flow on the right hand side of the figure consists of a DAS operation *PersonDAS.selectPerson* (3) that selects a row from table *Person* using certain filter criteria. If the result set is empty, a new row will be inserted into table *Person* by the DAS operation *PersonDAS.insertPerson* (5). Otherwise the retrieved row in table *Person* is updated by the DAS operation *PersonDAS.updatePerson* (4). Finally a row in table *Delivery* is updated by the DAS operation *DeliveryDAS.updateDelivery* (6).

All activities in a process flow instance are running in a transaction[121]. Consider two process instances *p1* and *p2* running through the main process. *P1* inserts a new row into table *Delivery* by performing the DAS operation *DeliveryDAS.insertDelivery* (1). At the same time *p2* updates a row into table *Person* by performing the DAS operation *PersonDAS.updatePerson* (4). Thus DAS operation *DeliveryDAS.insertDelivery* (1) holds table *Delivery* and DAS operation *PersonDAS.updatePerson* (4) holds table *Person*. As a result, the DAS operation *DeliveryDAS.updateDelivery* (6) cannot be executed because DAS operation *DeliveryDAS.insertDelivery* (1) holds table *Delivery*. Likewise, the DAS operation *PersonDAS.insertPerson* (2) cannot be executed because *PersonDAS.updatePerson* (4) holds table *Person*. This is the classic deadlock situation. In the figure, this deadlock situation is displayed by the intersecting arrows on the left hand side and, concomitantly, the non-intersecting arrows on the right hand side.

Without our persistent data access flow concept, analysts cannot solely focus on the persistent data access activities of the process, but must consider many other concerns at the same time. Therefore, especially if a large number of different types of activities is used in a flow model, manual deadlock detection will be an exhaustive and time-consuming task. Our approach is to provide a specific persistent data access flow that enables data analysts to focus only on the relevant information helpful for detecting deadlocks. In particular, our approach supports a visual solution to already eliminate potential deadlock risks at the modeling level. The same can be assessed for any other manual data analysis task in process-driven SOAs. Furthermore, on top of our approach, common deadlock detection techniques (such as[28,92,145]) can be performed.

Automatic deadlock detection with persistent data access flows In some cases, we want to go beyond manual data analysis in process-driven SOAs. The persistent data

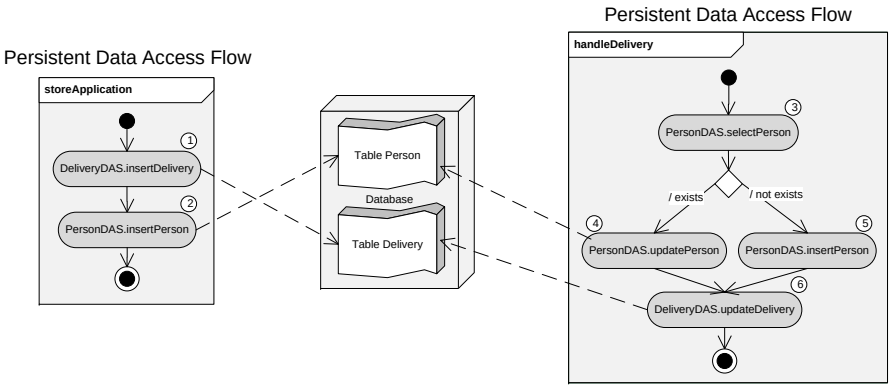


Figure 5.3: Motivating Example for Manually Detecting Potential Deadlock Risks

access flows enable us to easier implement algorithms for static deadlock detection in distributed database systems: As explained in the example above (see Figure 5.3), a deadlock can occur, when data resources in different persistent data access flows are accessed in a different order. Thus in order to detect possible deadlocks, we need to check the order in which database tables are accessed in each of these persistent data access flows. For this, we need to consider the paths of all persistent data access flows of the process. Accordingly, a possible deadlock algorithm compares the order in which database tables are accessed in one path *p1* with the order in which tables are accessed in another path *p2*. This pair-wise comparison needs to be done for each possible path of the persistent data access flows of a process.

Problem & Solution: Design Weakness Detection

In process-driven SOAs, at first, DAS developers have to become acquainted with the process flows including business logic activities, transformation activities, and persistent data access activities. In particular, they need a general overview of the persistent data access flows of the process e.g. they need to know which tables are accessed by a certain DAS operation. These persistent data access flows are in particular important for developers who need to review the developed database transactions in case of troubleshooting or analysis of performance leaks.

Secondly, in Integrated Development Environments (IDE) such as Eclipse[127], it is possible to search for modules that invoke a certain DAS operation. However, in a process flow of different types of activities, to search for specific DAS operations can be a time-consuming task. Accordingly, in contrast to our approach, in common IDEs it is not possible to extract a list of DAS operations accessing a specific database table or database table column.

Thirdly, the persistent data access flow enables DAS developers to easily discover inefficient persistent data access flow. Figure 5.4 shows an example of such an inefficient

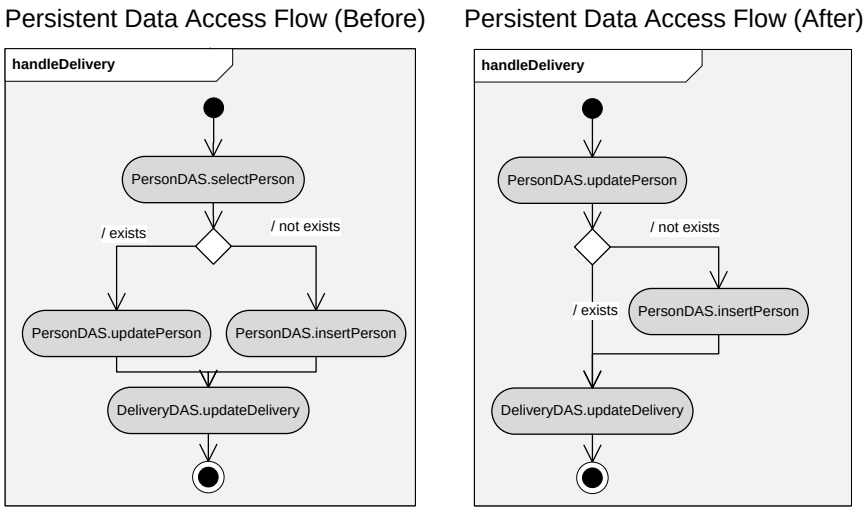


Figure 5.4: Motivating Example for Detecting Inefficient Persistent Data Access Flows

persistent data access flow before and after redesigning it. When we look at the flow on the left hand side of the figure, we can easily recognize that eliminating the DAS operation *DAS1.select* could reduce the number of statements during process execution. The reason for this is that the update operation *DAS1.update* anyway returns the number of updated data sets. After redesign, we can see the resulting flow on the right hand side of Figure 5.4. There are various performance measuring tools used to discover performance leaks at runtime. However we provide a visual approach to detect inefficient source code at the earliest possible state of the development process[28]. Our approach is not limited to the example above. It rather can be applied to solve many other types of structural problems in business processes.

Problem & Solution: Test Case Generation

One major task during testing a process is to check whether data is correctly stored and retrieved from a central storage. For this purpose, test cases have to be created, tested, and executed, and finally, the results need to be examined[51]. In the following, we concentrate on creating test cases at two different levels:

1. Test cases for single persistent data access activities: Each persistent data access activity will have to be checked whether data is correctly stored and accessed. Each persistent data access activity can be tested independently from the whole process. In order to create, test, and execute these test cases respectively, testers require necessary input and output data for each path of the process[112] (see Figure 5.5). In order to provide appropriate input and output data, they need the information which tables are accessed by a specific DAS operation. Our approach enables

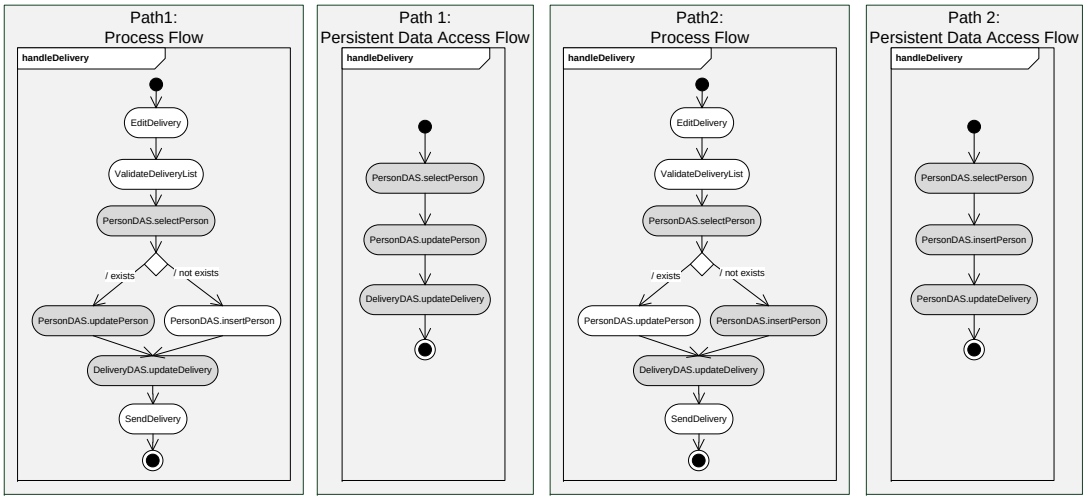


Figure 5.5: Motivating Example for Testing Persistent Data Access of a Process Flow

extracting persistent data access flows by different filter criteria, such as extracting persistent data access flows containing only those activities accessing a specific column of a database table.

2. Test cases for transactions: All persistent data access activities of a process are running within a transaction. For each possible path in this transaction, a test case has to be created, tested, and executed in order to verify the correctness of each path. Figure 5.5 exemplifies these different paths both of the process flow and of the associated persistent data access flow. The bold arrows mark a specific path within the process flow. In order to create, test, and execute cases for transactions, testers have to overview the overall persistent data access flows of a process. For this purpose, data test developers need a documentation of these modeled persistent data access flows. However, from our experience, in industry, persistent data access flows are not documented. Furthermore, even if such kind of documentation existed, the problem of updating this documentation in a timely manner would remain. "The only notable exception is documentation types that are highly structured and easy to maintain, such as test cases and inline comments"[77]. As our persistent data access flows follow the MDD[139] paradigm, there is no gap between specification and development. In particular, the effort to update the specification to be synchronized with the newly implemented data access activities is not necessary. The persistent data access flow concept provides such kind of documentation implicitly and thus enables testers to gain a better understanding of the persistent data access flows of a process. As shown in Figure 5.5, with our persistent data access flows, database testers can easily overview the persistent data access activity paths of a process. Finally, due to the improved persistent data access

documentation, we argue that using the persistent data access flow concept can decrease the participation of the different stakeholders during test case design.

Moreover, our persistent data access flow concept enables testers to locate errors more quickly when a specific test case asserting persistent data access fails[37]. Accordingly, testers will be able to verify the particular path of the flow and thus will more efficiently determine the failure reason e.g. if the failure is due to an error within the process, the test case or the provided input data. During the run, a log handler can log each persistent data access activity performed during the process. With this information, the error causing persistent data access activities can be easily retrieved by reconstructing the entire path of the persistent data access flow.

5.5 Model-Driven Solution: Specification, Integration, Extraction

In this section we prove the technical feasibility of our approach. Our model-driven solution is based on the view-based data modeling framework (see Section 4.2). The highly structured models are used as the modeling basis for extracting the flattened persistent data access flows. In the following we present the necessary steps to be taken in order to implement our persistent data access flow concept.

- **Specification** of persistent data access activities
- **Integration** of persistent data access activities with persistent data access implementation details
- **Extraction** of persistent data access flows from whole process flows

Specification

In Section 3.4 and Section 4.2 we already provided a general overview of the view-based modeling framework (VbMF) and view-based data modeling framework (VbDMF). Now, we present the newly defined VbDMF Flow View model that is used to specify the persistent data access activities of process flows. As shown in Figure 5.6, our VbDMF Flow View model is extended from the basic VbMF Flow View model.

The new VbDMF Flow View model consists of a separate persistent data access task *AtomicDASTask* extended from the basic *AtomicTask* of the VbMF Flow View. The VbMF *AtomicTask* class is a specialization of the VbMF *Task* class. The new *AtomicDASTask* allows stakeholders to being able to better focus on persistent data access in business processes. On the basis of this new simple Flow View model, we can link a business process activity with persistent data access implementation details. Below, we describe how stakeholders can associate each *AtomicDASTask* of the VbDMF Flow View with a corresponding *DAO operation* definition of the VbDMF DAO View.

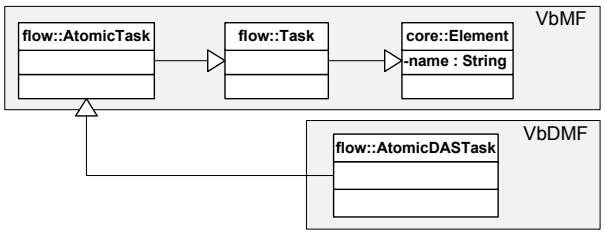


Figure 5.6: VbDMF Flow View Model

Integration

As explained before in Section 3.4, views can be enriched by the mechanism of view integration. Here, we enhance the concept of view integration by introducing view integration paths, which we use to trace implementation details of process activities among different views.

In Figure 4.2, we have basically outlined the (directed) view integration dependencies between the different VbMF/ VbDMF views. By the mechanism of view integration, persistent data access activities of a business process can be integrated with their persistent data access implementation details. In example, the Collaboration DAO Mapping View integrates the Collaboration View and the DAO View. The DAO View, in turn can integrate two views, namely the ORM View and the Data Object Type View. Finally, the ORM View can integrate the Physical Data View and the Data Object View. In order to visualize whether a process activity reads or writes from a certain database table, the Flow View needs to be integrated with the Physical Data View. However, as depicted in Figure 4.2, these two views are not connected directly. Thus, we have to establish an integration path between these two views.

To establish such an integration path from the source view i.e. the Flow View to the target view i.e. the Physical Data View, many views need to be connected. When integrating views to establish an integration path, the target view of the last view integration always becomes the source view of the next view integration. Within a view integration, we define the connection point in the source view as *integration start point* and the connection point in the target view as *integration end point*. In order to illustrate the concept of view integration paths, Figure 5.7 shows an integration path of four view integrations. The individual views are depicted in XMI notation.

- The first view integration combines the Flow View with the Collaboration DAO Mapping View. In this view integration, the entity *DASDelivery.insertDelivery* of type *AtomicDASTask* of the Flow View acts as integration start point S_1 and the entity *DASDelivery.insertDelivery* of type *AtomicDASTask* of the Collaboration DAO Mapping View acts as integration end point E_1) The Collaboration DAO Mapping View maps DAS operations to DAO operations e.g. it maps the DAS operation *DeliveryDAS.insertDelivery* of the Flow View to the DAO operation

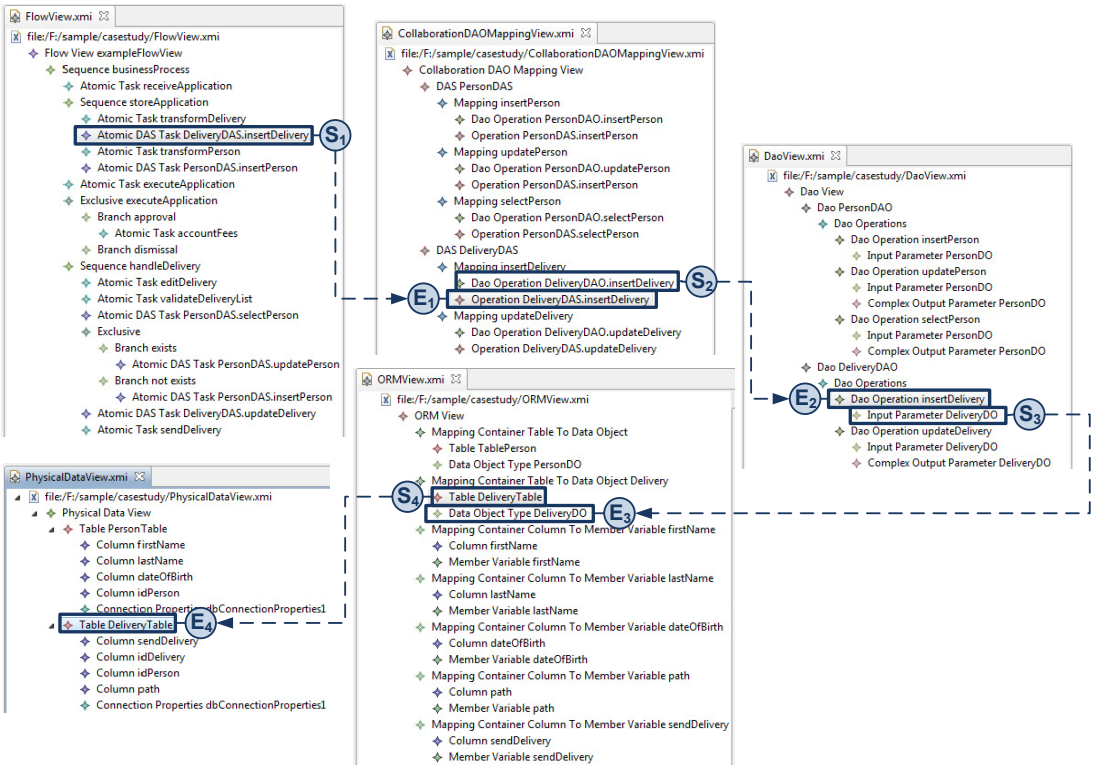


Figure 5.7: VbDMF Integration Path

DeliveryDAO.insert of the DAO View. Instead of using the Collaboration DAO Mapping View, the Flow View can also be integrated with the DAO View directly by using the VbMF/VbDMF's mechanism of view integration. However, in this case, as we use a name-matching algorithm for view integration (presented in [132], the DAO operations and the DAS operations would have to be named identically.

- The second view integration pair (S_2,E_2) combines the Collaboration DAO Mapping View with the DAO View. In this view integration, the *DeliveryDAO.insertDelivery* entity of type *AtomicDASTask* of the Collaboration DAO Mapping View is integrated with the DAO operation *DeliveryDAO.insertDelivery* entity of type *AtomicDASTask* of the DAO View.

- The third view integration combines the DAO View with the ORM View in order to get information about the associated database tables. Each *DAO Operation* of the DAO View contains *DAO Input Parameter Types* and *DAO Output Parameter Types*. Each parameter type can be mapped to corresponding *Data Object Types* of the ORM View. In our example, the entity *DeliveryDO* of type *DAO Input Parameter Type* of the DAO View can be mapped to the correspondent entity

DeliveryDO of type *DAO Input Parameter Type* of the ORM View. In this view integration, the entity *DeliveryDO* of the DAO View acts as integration start point S_3 and the entity *DeliveryDO* of the ORM View acts as integration end point E_3.

- The fourth view integration pair (S_4,E_4) finally combines the ORM View with the Physical Data View. In this final view integration, the *DeliveryTable* entity of type *Table* of the ORM View is integrated with the *DeliveryTable* entity of type *Table* of the Physical Data View. This is possible, because the ORM View maps *Tables* and *Table Columns* of the Physical Data View to *Data Object Types* and *Data Object Member Variables* of the Data Object Type View.

After illustrating the concept of view integration paths, we provide general definitions of the underlying terms.

Definition 1 *Let V_1 and V_2 be two views. If entity $S \in V_1$ matches entity $E \in V_2$ and entity $E \in V_2$ matches entity $S \in V_1$, then S will be defined as the* ***integration start point*** *and E will be defined as the* ***integration end point****. In this case, V_1 will be defined as the* ***source view*** *and V_2 will be defined as the* ***target view*** *of this view integration.*

Definition 2 *Let V_1 be a view, and M_1 be the model of V_1, such as $V_1 = instanceOf(M_1)$. An integration start point $S_1 \in V_1$* corresponds *to an integration end point $E_1 \in V_1$ when one of the following conditions is true:*

- $S_1 \equiv E_1$
- *E_1 is a super element of S_1*
- *Let MC_1 be a mapping container entity $\in M_1$ with S_1 $childOf(MC_1)$ and E_1 $childOf(MC_1)$.*

Definition 3 *Let V_i, $i \in 1..n$ be n views. A* ***view integration path*** *is a tuple of entity pairs $P(S_i, E_i | i = 1..n-1, S_i \in V_i, E_i \in V_{i+1})$ that meets the following conditions: $\forall$ $i \in 1..n-1$ $\exists$ $S_i \in V_i$ that matches an integration end point $E_i \in V_{i+1}$. Further, in a view integration path, each integration end point $E_i \in V_{i+1}$ corresponds to a new integration start point $S_{i+1} \in V_{i+1}$.*

An unsolved problem still is how to extract the persistent data access flows from the whole Flow View. In the following we present an algorithm calculating a flattened Flow View, defined as the DAS Flow View. Parts of this algorithm are the sub-algorithms implementing view integration as defined by the definitions 1–3 above.

Extraction

In the following we present our algorithm used to extract the DAS Flow View from the whole Flow View. Due to our model-driven view-based approach we can filter data access activities by specific filter criteria, such as tables, columns, DAOs, data objects etc. As a result our extracted persistent data access flows can contain only those activities accessing a specific table of a database. Before we define the algorithm to extract persistent data access flows from the whole process flow, in the context of VbDMF, we provide the following definition:

Definition 4: *The VbDMF DAS Flow View is an extraction of the VbDMF Flow View, accordingly, its tasks are of type* AtomicDASTask*. Hence, the DAS Flow View represents the persistent data access flows of a business process.*

Algorithms for global data flow analysis fall into two major classes: iterative algorithms and elimination algorithms[31]. In iterative algorithms, the equations are repeatedly evaluated until the evaluation converges to a fixed point. Elimination algorithms compute the fixed point by decomposition and reduction of the flow graph to obtain subsequently smaller systems of equations. We settled for a recursive elimination algorithm and present our simple recursive elimination algorithm *RecursiveClean* (Algorithm 5.1) to extract the DAS Flow View from the Flow View. Algorithm 5.1 contains the *MatchFilterCriteria* sub-algorithm (Algorithm 5.2 which is the heart of our recursive elimination algorithm *RecursiveClean*.

In the following we explain our recursive elimination algorithm *RecursiveClean* (Algorithm 5.1). The start *Task* of the Flow View is passed as mandatory input parameter to the algorithm. In addition, the optional input parameters *searchView* and *searchEntity* are passed to the algorithm in order to filter DAS operations by certain filter criteria. After executing the algorithm, the persistent data access flow contains only those DAS operations matching the entity *searchEntity* of the view *searchView*. In order to filter persistent data access flows by more than one filter criteria the algorithm can be performed repeatedly. If the input task of type *Task* has children, for all non-data-related entities, our recursive algorithm recursively will step into the different paths of the tree view. A task will be able to have children if it is of type *Sequence*, *Parallel*, *Exclusive* or *Branch*. For each non-data-related *childTask* of the current task, the algorithm calls itself recursively. As explained before, a task will be data-related, if it is of type *AtomicDASTask*. When stepping through a certain path, only the non-data-related leaf-tasks are removed by recursion from the Flow View. Per default, tasks of type *AtomicDASTask* must not removed, because they are part of our resulting DAS Flow View. Likewise, tasks such as *Sequence*, *Parallel*, *Exclusive* and *Branch* containing data-related entities must not removed as well, because they are also part of the resulting DAS Flow View. Algorithm *MatchFilterCriteria* (Algorithm 5.2) filters all data-related leaf-tasks of type *AtomicDASTask* provided that they do not match the filter criteria. In order to check whether the current leaf-tasks match the filter criteria, the algorithm *MatchFilterCriteria* tries to establish a view integration path to the entity *searchEntity* of the *searchView*.

```
   Input: Task task ∈ FlowView
   Input: View searchView
   Input: Entity searchEntity
 1 if (hasChildren(task)) then
 2   foreach Task childTask ∈ task.children() do
       /* only process non-data-related tasks                                  */
 3     if (!(childTask instanceof AtomicDASTask)) then
 4       recursiveClean(childTask);
 5       if (NOThasChildren(childTask)) then
 6         task.removeChild(childTask);
 7       end
 8     end
       /* only process data-related tasks                                      */
 9     else if (MatchFilterCriteria(childTask, searchView, searchEntity)) then
10       task.removeChild(childTask);
11     end
12   end
13 end
14 else if (!(task instanceof AtomicDASTask)) then
15   task = null;
16 end
17 else if (MatchFilterCriteria(childTask, searchView, searchEntity)) then
18   task = null;
19 end
```

Algorithm 5.1: RecursiveClean()

Hereby the current leaf-tasks act as integration points. If a view integration path is found, Algorithm 5.2 will return true.

```
   Input: Entity integrationStartPoint ∈ FlowView
   Input: View searchView
   Input: Entity searchEntity
 1 sourceView=FlowView;
 2 if (NOT(searchView == null)) then
 3   while (NOTsourceView.equals(searchView)) do
 4     targetView = getNextView(sourceView, searchView);
 5     integrationEndPoint = getIntegrationEndPoint(integrationStartPoint, targetView);
 6     if (NOT(searchView.equals(targetView))) then
 7       integrationStartPoint =
         RecursiveGetIntegrationStartPoint(integrationEndPoint, targetView);
 8     end
 9     sourceView = targetView;
10   end
11 end
12 return (RecursiveMatchEntity(integrationEndPoint, searchEntity));
```

Algorithm 5.2: MatchFilterCriteria()

The sub-algorithm *MatchFilterCriteria* algorithm (see Algorithm 5.2) is the heart of our implementation solution. It checks whether a certain process activity matches given

filter criteria by implementing the concept of view integration paths. Algorithm 5.2 consists of three basic functions:

- *getNextView(View sourceView, View targetView)* The function *getNextView* simply returns the next related view based on the *sourceView* and the *targetView*. The function returns the next view based on the view integration dependencies depicted in Figure 4.2. The contained function *NextEntity* is comparably simple, and thus is not further illustrated.

- *getIntegrationEndPoint(Entity startEntity, View targetView)* In order to connect a source view with a target view, the integration start point of the source view needs to be integrated with an integration end point of the target view. In our prototype implementation, the algorithm finds the corresponding integration end point in the target view based on name-matching[132]. As the name-matching algorithm (presented in[132]) is sufficient for our prototype implementation, we do not provide further implementations to find matching entities in the target view in this thesis.

- *RecursiveGetIntegrationStartPoint(Entity oldEntity, View sourceView)* Based on the integration end point of the previous view integration, this Algorithm 5.3 can calculate the integration start point of the next view integration. The target view of the last view integration becomes the source view of the next view integration. Thus, the old integration end point is in the same view as the new integration start point. Algorithm 5.3, the heart of our integration path calculation, requires the simple recursive sub-algorithm *RecursiveMatchEntity* (Algorithm 5.4) to check whether the integration end point of the target view contains or matches given filter criteria.

Three parameters are passed to Algorithm 5.2:

- The persistent data access activity *integrationStartPoint* of the Flow View, which is the activity to be checked for a filter criteria match.

- The filter criteria to be checked contained in the view *searchView*.

- The entity *searchEntity* representing the filter criteria.

As defined above, a view integration path consists of pairs of each a integration start entity and an integration end entity. Each integration start entity belongs to the source view and each integration end entity belongs to the target view. Accordingly, at first, a variable *sourceView* is initialized within the Flow View. As long as the current *sourceView* does not equal the *searchEntity* entity of the *searchView*, the functions *getNextView*, *getIntegrationEndPoint*, and *RecursiveGetIntegrationStartPoint* are invoked. The function *RecursiveGetIntegrationStartPoint* is invoked as long as the variable *targetView* does not equal the variable *searchView*.

In the following we describe the algorithm *RecursiveGetIntegrationStartPoint* (Algorithm 5.3) used to get the integration start point of the current view integration. The

```
   Input: Entity currentEntity
   Input: View targetView
 1 integrationEndPoint=GetIntegrationEndPoint(currentEntity, targetView);
 2 if NOT(integrationEndPoint == null) then
 3   return integrationEndPoint;
 4 end
 5 else
 6   if (hasChildren(currentEntity)) then
 7     foreach (Entity childEntity ∈ entity.children()) do
 8       return RecursiveGetIntegrationStartPoint(childEntity, targetView);
 9     end
10   end
11   else
12     Entity parent = getParent(currentEntity);
13     if (parent instanceof MappingContainer) then
14       foreach (Entity childEntity ∈ parent.children()) do
15         if (NOT (childEntity.equals(currentEntity))) then
16           return childEntity;
17         end
18       end
19     end
20   end
21 end
22 return NULL;
```

Algorithm 5.3: RecursiveGetIntegrationStartPoint()

input parameters *currentEntity* and *targetView* are passed to Algorithm 5.3. According to Definition 2, there are three possibilities how to find an integration start point for the next view: The new integration start point either is the last integration end point, the new integration start point is part of a *Matching Container*, or the new integration start point is a child of the last integration end point. According to the first possibility, the function *GetIntegrationEndPoint* checks whether the current integration start point of the *currentView* matches a corresponding integration end point in the target view. If the integration start point matches a corresponding integration end point, a new integration start point will be found. Otherwise, the new integration start point is either part of a *Matching Container* or it becomes a child entity of the current entity. In both cases, the *RecursiveGetIntegrationStartPoint* algorithm invokes itself recursively to check whether the new integration start point matches an integration end point.

Finally, the simple recursive algorithm *RecursiveMatchEntity* (Algorithm 5.4) is invoked in order to check whether the integration point in the target view matches the given filter criteria. For this purpose, two parameters are passed to Algorithm 5.4. Firstly, the entity *currentEntity* is to be checked against certain filter criteria. Secondly, the entity *searchEntity* specifies this filter criteria. The algorithm firstly checks whether the entity *searchEntity* equals *currentEntity* by name-matching. If this is true, the business process activity will match the filter criteria. If *currentEntity* has children, for each child, the algorithm will invoke itself recursively.

```
   Input: Entity currentEntity ∈ ViewcurrentView
   Input: Entity searchEntity ∈ ViewcurrentView
1  if (currentEntity.equals(searchEntity)) then
2    return TRUE;
3  end
4  if (hasChildren(currentEntity)) then
5    foreach Entity childEntity ∈ entity.children() do
6      RecursiveMatchEntity(childEntity);
7    end
8  end
9  return FALSE;
```

Algorithm 5.4: RecursiveMatchEntity()

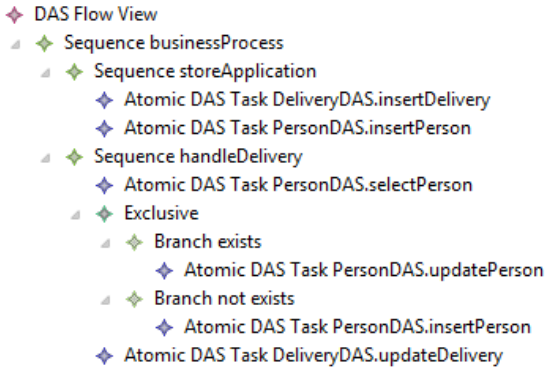
To conclude, our recursive elimination algorithm can be reused for extracting other views such as for extracting all service operations from the Flow View.

5.6 Applicability of the Algorithms & Tooling

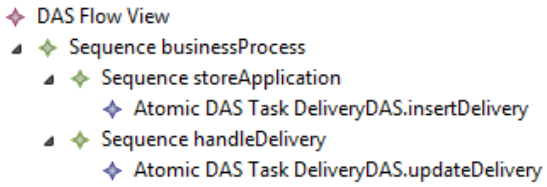
In this section we show the applicability of the algorithms above and present a suitable tooling. For this, we exemplary apply our recursive elimination algorithm to the business process of the case study presented in Section 4.4. We illustrate how our algorithms can be applied to extract both simple and filtered persistent data access flows from the business process flow.

Extract simple persistent data access flows When stakeholders want to test persistent data access in process driven SOAs, as shown in Section 5.4, they need a documentation about the persistent data access activities in the business process. Our simple persistent data access flow provides such a documentation. In the following we apply our algorithms to extract the DAS Flow View from the Flow View modeling the business process flow shown in Figure 5.2. For this purpose, we invoke the recursive elimination algorithm *RecursiveClean* (Algorithm 5.1) with the start *Task* of the Flow View. The resulting Flow View of this algorithm is a DAS Flow View that only contains persistent data access activities. We invoke the algorithm with the *null* value for the input parameters *searchView* and *searchEntity*. Figure 5.8(a) shows the XMI notation of the extracted DAS Flow View after invoking the algorithm. The resulting DAS Flow View consists of two simple persistent data access flows, represented as *Sequence* elements.

Extract filtered persistent data access flows In case a deadlock occurs, data analysts want to check the business process for structural problems. For this purpose, they can extract all the persistent data access activities that read or write from a specific database table. In our example the function *MatchFilterCriteria* of Algorithm 5.1 filters all persistent data access activities that do not access a specific table *DeliveryTable*. To establish this, we set the input parameters *searchView* and *searchEntity* to the values *Physical Data View* and *Table* respectively. In addition, we set the attribute *Table.name*



(a) DAS Flow View



(b) DAS Flow View (Table Delivery)

Figure 5.8: XMI Notation of a Simple and Filtered DAS Flow View

to *DeliveryTable*. By view integration, we can filter those persistent data access activities not accessing the specific table *DeliveryTable*. In Figure 5.7, we illustrate the view integration path between the Flow View and the Physical Data View. In order to establish the view integration path, each persistent data access activity has to be integrated with the DAO View in order to get the corresponding DAO operation. The *Input Parameter Data Object Types* and the *Output Parameter Data Objects Types* of the *DAO operation* are then integrated with the *Data Object Types* of the ORM View. Within the ORM View each *Data Object Type* is mapped to a specific *Table*. Finally, we integrate the *Table* entity of the *ORM View* with the *Table* entity of the Physical Data View. Finally, the algorithm *RecursiveMatchEntity* (Algorithm 5.4) checks each current entity in the Physical Data View against the *DeliveryTable* entity. If the current entity matches the *DeliveryTable* entity, the related persistent data access activity will be part of the resulting persistent data access flow. Otherwise the concerned persistent data access activity is filtered from the Flow View. The resulting extracted DAS Flow View is shown in Figure 5.8(b). As a result, the DAS Flow View contains two persistent data access flows (represented as *Sequences*) of the whole business flow. Both resulting filtered persistent data access flows are characterized by containing only those persistent data access activities accessing table *DeliveryTable*.

In the following, we give a few more use case examples fulfilled by stakeholders developing and maintaining applications in large-scale enterprises. Whether a certain use case occurs depends e.g. on the quality of the underlying business process and non-

functional requirements e.g. the availability of external dependencies such as service providers and databases. These use cases mainly result from our study of analyzing persistent data access in service-oriented environments in a large enterprise and secondly from analyzing literature in this field. They demonstrate how our persistent data access flows can be applied to specific analysis problems. Each of these use cases extracts a persistent data access flow from the whole process flow by different filter criteria.

- In case a deadlock occurs, in addition to selecting all persistent data access activities accessing a specific database table, data analysts can further flatten the resulting persistent data access flow. In example, they can extract all the persistent data access activities from the business process that read or write from a specific column of a database table.

- In case a specific database fails, stakeholders such as DAS developers need a documentation of which business process activities access a specific database. For this purpose, they need to extract all the persistent data access activities that read or write data from a specific database url. In order to establish this, in addition to the previous four view integrations, the Physical Data View needs to be integrated with the *Database Connection View*.

- Let us consider the case a certain service provider is shut down for any reason. Then, stakeholders such as system architects need to determine the business process activities invoking services running on the failed service provider. For this purpose, stakeholders can extract only those persistent data access activities from the whole process flow which run on a certain URI *Service.Uri.name*. In order to establish this, the algorithm *MatchFilterCriteria* integrates the DAS Flow View with the *Collaboration View*.

In order to demonstrate applicability of our model-driven solution, Figure 5.9 shows a suitable tooling. The tooling shall support stakeholders both to view the persistent data access flows of a business process and to trace persistent data access details of the process activities. Top middle in the figure, the relationships between the process flow, the simple persistent data access flow, and the filtered persistent data access flow are shown. In example, the dotted lines illustrate the view integration path of the persistent data access activity *DeliveryDAS.insertDelivery* with its related views. In the figure, each view is labeled with a number that has a corresponding number in the descriptions below.

1. Persistent Data Access Flow Filter Criteria: Stakeholders can select filter criteria. Based on the filter criteria, the recursive elimination algorithm *RecursiveClean* (Algorithm 5.1) is invoked with certain input parameter values. In the example we select *PhysicalDataView.table.name=DeliveryTable* to filter all persistent data access activities from the flow that do not match the filter criteria. As soon as the search button is pressed, as explained before, Algorithm 5.1 is invoked with the input parameter values *Physical Data View*, *Table*, and *DeliveryTable*.

2. Persistent Data Access Flow Filter Results: In this window, stakeholders can view the resulting persistent data access activities matching the filter criteria. In our example, two persistent data access activities have been found. The first persistent data access activity *DeliveryDAS.insertDelivery* is part of the sub process *storeApplication* The second persistent data access activity *DeliveryDAS.updateDelivery* is part of the sub process *handleDelivery*.

3. The Flow View shows the whole business process.

4. The DAS Flow View contains a simple business process that only consists of the persistent data access activities.

5. The filtered persistent DAS Flow View shows a filtered business process that only consists of the persistent data access activities reading or writing from Table *DeliveryTable*.

6. Other related views: Stakeholders can view persistent data access details of persistent data access activities such as physical storage tables, database connections, object-relational mappings, and data access object types. In Figure 5.9, the implementation details of the business process activity *DeliveryDAS.insertDelivery* are labeled in yellow. The dotted lines exemplify a view integration path of the persistent data access activity *DASDelivery.insertDelivery* with its related views. So, the VbMF Flow View can integrate each *AtomicTask* of the Flow View with further service operation definitions from the Collaboration View. As an extension of the VbMF Flow View, the VbDMF Flow View describes a flow of process activities consisting of activities invoking DAS operations. The Collaboration View specifies the service operation definitions of the DAS operations. The DAO Collaboration Mapping View maps data access service (DAS) to underlying data access object (DAO) definitions. Furthermore, the DAO View models the underlying DAO operations of the DAS operations. The Physical Data View integrates the Database Connection View in order to get the *DBConnectionProperties* details of the underlying RDBMS such as url, user, password etc. The integration of the Physical Data View with the Database Connection View is possible by the entity *Table* of the Physical Data View incorporating a reference to the *dbConnectionProperties* entity acting as integration point.

5.7 Evaluation

In this section we want to discuss both the correctness and complexity of the presented algorithms.

Correctness Here, we discuss the correctness of the algorithm *MatchFilterCriteria* (Algorithm 5.2) using induction. The *MatchFilterCriteria* algorithm is the heart of our

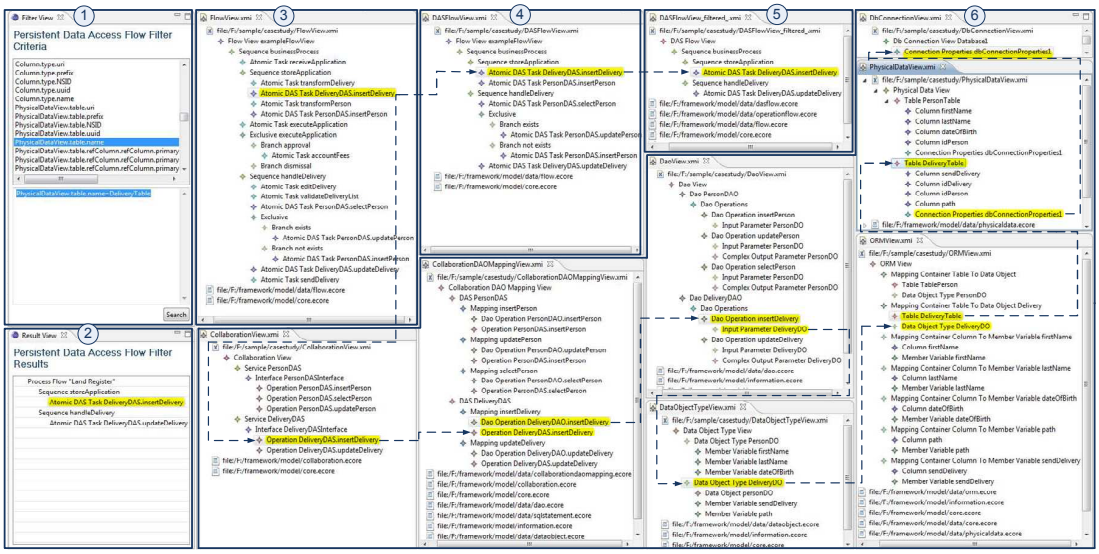


Figure 5.9: Tooling for Tracing Persistent Data Access in Process-Driven SOAs

recursive elimination algorithm that is used to implement our view integration path concept.

Hypothesis: *Let VT_i be the i-th target view and VS_{i+1} be the (i+1)-th source view of a certain view integration path. The algorithm will be correct, if the target view VT_i equals the source view VS_{i+1} $\forall$ $2 < i < n$.*

Algorithm 5.5 illustrates a reduced *MatchFilterCriteria* algorithm that contains the relevant lines of the while loop to prove the hypothesis. In this reduced *MatchFilterCriteria* algorithm we use the following variables: VS_i denotes the i-th source view of a view integration path, VT_i denotes the i-th target view of a view integration path. Accordingly, $S_i \in VS_i$ denotes the integration start point of the i-th view integration and $E_i \in VT_i$ denotes the integration end point of the i-th view integration.

```
1 while (NOT VS_i.equals(searchView)) do
2     targetView = getNextView(VS_i, searchView);
3     E_i = getIntegrationEndPoint(S_i, targetView);
4     if (NOT(searchView.equals(VT_i))) then
5         S_{i+1} = RecursiveGetIntegrationStartPoint(E_i, VT_i);
6     end
7     VS_{i+1} = VT_i;
8 end
```

Algorithm 5.5: Reduced MatchFilterCriteria Algorithm

Let i be the number of while loop cycles of Algorithm 5.5. The number of while loop cycles equates the number of views in the view integration path. $\forall\ 0 < i < 2$ the hypothesis is false, because a view integration path must have at least two views in order to fulfill the hypothesis:

1. $i = 1 : S_1 \in VS_1, T_1 = null$
2. $i = 2 : S_1 \in VS_1, T_1 \in VT_1$

Base Case: $i = 3 : S_1 \in VS_1,\ T_1 \in VT_1,\ S_2 in VS_2,\ VS_2 = VT_1,\ T_2 \in VT_2$

Inductive Step: *Let* $VS_i = VT_{i-1}$ *be true* $\forall\ 2 < i < n$*:*
$S_1 \in VS_1,\ T_1 \in VT_1,\ ..,\ S_{n-1} \in VS_{n-1},\ T_{n-1} \in VT_{n-1},\ S_n \in VS_n,\ VS_n = VT_{n-1},\ T_n \in VT_n$.

Now, we show that the hypothesis is true $\forall\ 2 < i < n + 1$:

$S_1 \in VS_1,\ T_1 \in VT_1,\ ..,\ S_{n-1} \in VS_{n-1},\ T_{n-1} \in VT_{n-1},\ S_n \in VS_n,\ VS_n = VT_{n-1},\ T_n \in VT_n,\ S_n \in VS_n,\ T_n \in VT_n,\ S_{n+1} \in VS_{n+1},\ VS_{n+1} = VT_n,\ T_{n+1} \in VT_{n+1}$
From this it follows that $\forall\ 2 < i < n + 1$: $VS_{i+1} = VT_{(i+1)-1} = VT_i$. Hereby we have proven that our hypothesis is true.

Complexity: In the following we quantitatively measure the complexity of the presented algorithms using the Big O notation. We evaluate each of our algorithms separately before we will derive the overall performance from the parts.

Formally, the algorithm $f(n)$ will be equivalent to $O(g(n))\ \forall\ n > 0$, if there exists a constant $c > 0$, such that $f(n) = c * g(n)$. Table 5.1 summarizes the complexity of the presented algorithms. The complexity of each algorithm is presented in a separate sub table. The presentation order of the sub-algorithms is bottom-up such that the complexity of the sub-algorithms are presented before the parent *RecursiveClean* algorithm. In each sub table, the line number, the complexity of each line, and the maximum number of invocations are displayed. Firstly, in Table 5.1(a) and Table 5.1(b) the complexity of the algorithms *RecursiveGetIntegrationStartPoint* and *RecursiveMatchEntity* are illustrated. As these two algorithms are invoked by algorithm *MatchFilterCriteria*, next, Table 5.1(c) shows the complexity of the algorithm *MatchFilterCriteria*. Finally, Table 5.1(d) displays the complexity of the algorithm *RecursiveClean* which invokes the algorithm *MatchFilterCriteria*. In the sub tables, we use the following literals: v to refer to the number of views in a view integration path, n to denote the number of elements within a view, and the constant k to denote the number of child elements within an integration element within a view. For example, The input parameter *DeliveryDO* of the DAOView is a child entity of the *DeliveryDAO.insert* integration point. The number of persistent data access activities within the Flow View is denoted by d.

Table 5.1(a) depicts the complexity of the recursive algorithm *RecursiveGetIntegrationStartPoint* (Algorithm 5.3). The algorithm *RecursiveGetIntegrationStartPoint* will check each entity of the view whether it matches the current entity *currentEntity*. Thus,

Table 5.1: Algorithm Complexity

(a) Complexity of Algorithm RecursiveGetIntegrationStartPoint (Algorithm 5.3)

Line #	Line of Algorithm	Complexity of Line	max. # of Invocations
1	$integrationEndPoint$=$GetIntegrationEndPoint(currentEntity, targetView)$	O(n)	k
2	If(NOT($integrationEndPoint == null$))	O(1)	k
3	Return(integrationEndPoint)	O(1)	k
4	Else		
5	If (($hasChildren(currentEntity)$))	O(1)	k
6	ForEach((Entity $childEntity \in entity.children()$))	O(1)	k
7	RETURN ($RecursiveGetIntegrationStartPoint(childEntity, targetView)$)	O(1)	k
8	Else		
9	Entity $parent = getParent(currentEntity)$	O(1)	k
10	If(($parent$ instanceof $MappingContainer$))O(1)	k	
11	ForEach((Entity $childEntity \in parent.children()$))	O(1)	k
12	If((NOT (childEntity.equals(currentEntity))))	O(1)	k
13	RETURN childEntity	O(1)	k
14	RETURN NULL	O(1)	k

(b) Complexity of Algorithm RecursiveMatchEntity (Algorithm 5.4)

Line #	Line of Algorithm	Complexity of Line	max. # of Invocations
1	If (($currentEntity.equals(searchEntity)$))	O(1)	k
2	ReturnTRUE	O(1)	1
3	If ($hasChildren(currentEntity)$)	O(1)	k
4	Else		
5	ForEach(Entity $childEntity \in entity.children()$)	O(1)	k
6	RecursiveMatchEntity(childEntity)	O(1)	k
7	ReturnFALSE	O(1)	1

(c) Complexity of Algorithm MatchFilterCriteria (Algorithm 5.2)

Line #	Line of Algorithm	Complexity of Line	max. # of Invocations
1	sourceView=FlowView	O(1)	1
2	if (NOT searchView==null))	O(1)	1
3	while (NOT sourceView.equals(searchView))	O(1)	v
4	targetView = getNextView(sourceView; searchView)	O(1)	v-1
5	integrationEndPoint = getIntegrationEndPoint(integrationStartPoint, targetView)	O(n)	v-1
6	if (NOT(searchView:equals(targetView))) then	O(1)	v-1
7	integrationStartPoint = RecursiveGetIntegrationStartPoint (integrationEndPoint,targetView)	O(1)	v-2
8	sourceView=targetView	O(1)	v-1
9	return (RecursiveMatchEntity(integrationEndPoint, searchEntity))	O(1)	1

(d) Complexity of Algorithm RecursiveClean (Algorithm 5.1)

Line #	Line of Algorithm	Complexity of Line	max. # of Invocations
1	If (($hasChildren(task)$))	O(1)	n
2	ForEach(Task $childTask \in task.children()$)	O(1)	n
3	If((!($childTask$ instanceof $AtomicDASTask$)))	O(1)	n
4	$recursiveClean(childTask)$	O(1)	n-d
5	If(($NOThasChildren(childTask)$))	O(1)	n-d
6	$task.removeChild(childTask)$	O(1)	n-d
7	ElseIf(($MatchFilterCriteria(childTask, , searchView, searchEntity)$))	O(d)	d
8	$task.removeChild(childTask)$	O(1)	d
9	ElseIf ((!($task$ instanceof $AtomicDASTask$)))	O(1)	n-d
10	$task = null$	O(1)	n-d
11	ElseIf(($MatchFilterCriteria(childTask, , searchView, searchEntity)$))	O(d)	d
12	$task = null$	O(1)	d

the function *RecursiveGetIntegrationStartPoint* is of linear complexity $O(n)$ and is invoked at most m times, whereas m corresponds to the number of child elements of entity currentEntity. However, the number of child elements m is not dependent on the number of process elements, because it is a constant factor. Therefore, the next statements are also of constant complexity. As a result, the overall performance of the algorithm *RecursiveGetIntegrationStartPoint* is linear.

Table 5.1(b) summarizes the complexity of the algorithm *RecursiveMatchEntity* (Algorithm 5.4). As an entity has a constant number of child entities e.g. the entity *Table* has a constant number of *Columns*. Therefore, the algorithm *RecursiveMatchEntity* is of constant complexity.

Table 5.1(c) illustrates the complexity of Algorithm 5.2. The function *getNextView*, that is not further specified, is of constant complexity, because it simply returns the next view by the current view. In contrast to the function *getNextView*, the function *getIntegrationEndPoint* is dependent on the number of process elements within a view. The function *getIntegrationEndPoint* determines a matching element (the integration end point) in the target view, that is the integration end point. Thus, the response time of this function grows linearly with the number of view elements. The function *RecursiveMatchEntity* checks whether the current entity matches the filter criteria. This function is also of constant complexity. As a result, Algorithm 5.2 has a linear overall performance.

Table 5.1(d) shows the complexity of the algorithm *RecursiveClean* (Algorithm 5.1). Each line in the algorithm is invoked linearly with the number of process activities in the business process. In particular, line 7 and 11 are invoked linearly with the number of persistent data access activities of the business process. The lines 8 and 12 will only be invoked, if the persistent data access activities do not match the filter criteria. Thus, the overall performance of our recursive elimination algorithm *RecursiveClean* for $d > 0$ is $O(d^2)$. For $d = 0$, the worst case response time of the recursive elimination algorithm *RecursiveClean* grows solely linear with the number of process activities $O(n)$. Again, d is the number of persistent data access activities of the Flow View.

In the following we summarize the resulting complexity of the algorithms.

- RecursiveGetIntegrationStartPoint: $k * O(n) + 11 * k * O(1) \equiv O(n)$
- RecursiveMatchEntity: $4 * k * O(1) + 2 * O(1) = (2 * (2k + 1)) * O(1) \equiv O(1)$
- MatchFilterCriteria: $(v-1)*O(n)+v*O(1)+2*(v-1)*O(1)+(v-2)*O(1)+3*O(1)$ $= (v - 1) * O(n) + (4v - 1) * O(1) \equiv O(n)$
- RecursiveClean:
  for $d = 0$: $3n*O(1)+5*(n-d)*O(1)+2d*O(d)+2d*O(1) \equiv 3n*O(1)+5*(n)*O(1) \equiv O(n)$

  for $d = n$: $3n*O(1)+5*(n-d)*O(1)+2d*O(d)+2d*O(1) \equiv 3d*O(1)+2d*O(d)+2d*O(1)$ $= 5d * O(1) + 2d * O(d) \equiv O(d^2)$

for $0 < d < n$: $3n * O(1) + 5 * (n - d) * O(1) + 2d * O(d) + 2d * O(1) \equiv 3 * O(n) + 5 * O(n - d) + 2 * O(d) + 2 * O(d^2) \equiv O(d^2)$

Today, XML is a popular standard data exchange format. Thus, in literature, there is a variety of more efficient XML structural matching techniques[4,19]. By using these structural matching techniques, the worst case complexity of our recursive elimination algorithm $O(d^2)$ for $d > 0$ could be reduced. However, the aim of this section is to quantitatively show the feasibility and applicability of our approach, which has been achieved well.

5.8 Discussion

Different stakeholders such as data analysts, DAS developers, and database testers have different requirements to a software system. According to the pattern of separation of concerns[45], appropriate views must be provided to the different stakeholders. However, in addition to these views, read-only sub-views extracted from these rich views can facilitate tasks such as developing, and testing. Thus, besides view model extension and view integration, we introduced a further mechanism in order to generate a resulting view: The mechanism to extract views from existing views. In this connection we need to distinguish between editable and read-only views. The DAS Flow View is an example of such a read-only view. The DAS Flow View cannot be specified at modeling time, because usually connections have to be modeled in the context of the whole business flow. Hence, these extracted views are typically read-only views. Moreover, our DAS Flow View can be generated from the Flow View on the fly. Thus the DAS Flow View does not have to be stored after adapting the corresponding Flow View. This concept is comparable to the view concept in database theory: A database view can output data stored in one more database tables. When data in one of these database tables changes, the database view can output the updated data by accessing them through the tables. The disadvantage of this on-the-fly-generation is that the generation procedure needs to be performed each time when selecting the DAS Flow View.

To the best of our knowledge, up-to-now these persistent data access flows are not used to solve analysis, development, and testing problems, yet. In this chapter, our goal is to present a visual solution for a series of persistent data access problems. Accordingly, the specified scenarios in Section 5.4 are just examples of how our approach can be applied. Furthermore, we discover deadlocks by ensuring whether the DAS operations are properly designed. However with our approach we do not claim to discover a new approach for detecting deadlocks. The potential deadlock cause of two process-instances invoking intersecting DAS operations presented, is just one of several possible causes for a deadlock. Other mistakes such as an incorrect transaction handling or database configuration can also increase the probability of a deadlock. Instead, our persistent data access flow shall ease both manual and automatic deadlock detection in a complex process model. On top of our approach existing data analysis solutions such as deadlock detection techniques can be applied.

In the following we shortly state how our approach reduces the complexity of the process in the context of the three presented use cases. Hereby we use the definitions for process complexity specified in [21]. Four main metrics can be identified to measure the complexity of a process: activity complexity, control flow complexity, data-flow complexity, and resource complexity. The activity complexity of the process simply calculates the number of activities a process has. The control flow behavior of a process is affected by process constructs such as splits, joins, loops, and ending and starting points. The data-flow and resource complexity perspectives measure the complexity of data structures and the diversity of resources respectively. With our approach, according to the concept of separation of concerns, we could reduce the number of activities of a flow. We achieved this by extracting persistent data access flows consisting of simply data access activities. Thus we reduced the activity complexity of the process. Furthermore in the database testing use case, we resolved one complex problem into a number of simpler problems by extracting the data paths from a whole process flow. In this use case we could also decrease the control flow complexity to a minimum value. This is because a data path contains no switch constructs. By our filtering mechanism, we could also reduce the data-flow complexity and resource-complexity of business processes.

5.9 Summary

Process flows contain different types of activities such as business logic activities, transformation activities, and persistent data access activities. When the number of activities grows, focusing on special types of activities of the process flow such as the persistent data access activities is a time-consuming task. In this work we presented a view-based, model-driven solution extracting persistent data access flows from the whole process flow. By using these persistent data access flows, different stakeholders such as data analysts, DAS developers, and database testers can focus on the persistent data access activities of the process flows and to solve structural problems in business processes. We illustrated how our tailored DAS Flow View concept can improve data analysis, development, and testing by presenting selected use cases. Each of these use cases is an example of how persistent data access flows can increase efficiency and decrease the time to solve certain problems at the earliest state of development. Our DAS Flow View can be further tailored by different filter criteria such that the flow contains only those persistent data access activities reading or writing data from a specific database table. We have demonstrated the applicability of our approach by a suitable tooling. Furthermore, we have evaluated the feasibility by showing the correctness and complexity of the presented algorithms. Apart from focusing on the persistent data access activities, our approach can be generally applied to focus on any particular parts of the business process in a process-driven SOA.

CHAPTER 6

Reusable Architectural Decision Model for Model and Metadata Repositories

In this chapter we describe reusable knowledge in form of reusable architectural decisions for system architects in setting-up, planning, and developing model and metadata repositories, as well as the main decision drivers. For each decision we present recommendations which alternative to choose depending on certain requirements and boundary conditions. Thus, our approach basically aims at decreasing the costs and impact of making wrong decisions related when setting-up model and metadata repositories. In addition, it can be used as a lightweight approach to architecture documentation: If the reusable architectural decision model is used to make decisions, only a reference to the decision model will be needed to document an architectural decision instead of documenting the whole decision as well as the rationale. Our research results are based on field notes and observations from own model repository projects, a detailed analysis of existing model repository projects (both open source and commercial), and interviews and discussions with other model repository developers.

This chapter is structured as follows: Firstly, in Section 6.1 we motivate our approach. Next, in Section 6.2, we both define reusable architectural decision models (RADM) forming the background for this chapter. Section 6.3 provides detailed specifications of the architectural decisions and describes the dependencies between them. In Section 6.4, the applicability of the RADM is illustrated by a case study. Finally Section 6.5 sums up this chapter.

6.1 Motivation

When setting up model and metadata repositories, repositories should be optimized for the kind of modeling artifacts they store and the task they should fulfill. For instance, custom, model-aware queries should be provided that are simplified or more powerful compared to standard queries, such as SQL queries, because they can make use of the information in the modeling artifacts. In addition, model and metadata repositories are often realized on top of existing basic technology such as databases, but it is not enough to simply store the models in and retrieve them from such a basic technology. Therefore, a lot of architectural decisions have to be made. Some of the resulting decisions might be intuitively decided in a suitable way by system architects. However, other decisions might be skipped or decided in a non-optimal way because of missing knowledge of alternatives and consequences.

In this chapter we present our reusable architectural decision model (RADM), documenting the decisions for setting up model and metadata repositories. This RADM can be instantiated for a concrete system as shown in the case study Section 6.4.

6.2 Background

According to Jansen and Bosch[62] "software architecture is a set of principal design decisions". During a software system's design phase, system architects have to make numerous decisions for organizational and business issues, for matters of broad and detailed design, and for technologies[148]. We will refer to a design decision using the term *architectural decision*, if it meets the following conditions: Firstly, it affects either the architecture of a system or the role of the system architect, and secondly, the system architects of the system see those decisions as *principal* decisions. The main argument for using architectural decision modeling is that such principal decisions should not get lost.

Architectural decision models are used to document architectural decisions[62,73,134]. These architectural models capture selected decision options and justifications for these decisions. However, from our experience, in industry, stakeholders often do not attach great value to documenting decisions, and, if it is performed at all, documentation will usually be done in retrospect. Thus, architectural decision models cannot solve the problem of lacking documentation for architecture decisions.

Reusable architectural decision models proposed by Zimmermann et al.[147,148] focus on solving these problems: "A *reusable architectural decision model* enhances the basic decision model by steering the architectural decision making activities[148]". Reusable decision models are closely related to software pattern concepts[50]. For instance, Zimmermann et al.'s approach applies the reusable architectural decision models for pattern selection.

In this chapter, we describe a reusable architectural decision model for model and metadata repositories. Each *architectural decision* is characterized by a *decision name*. In this model, the decisions either have a number of *alternatives* or *options* between

which to decide. Some alternatives or options have *variants*, which can be selected. For each decision, we describe the *forces* or *decision drivers* that must be considered, when selecting an alternative or option. Usually, the different alternatives and options have different *consequences* with regard to the forces. For each alternative and option, we describe a few *known uses*. Finally, decisions have *relationships* with other decisions. For instance, a decision will be able to be a follow-on decision to another decision, if a specific alternative or option is chosen.

6.3 Architectural Decisions

In this section, we describe architectural decisions architects must make for planning, setting-up, developing, and installing model and metadata repositories. In particular, we focus on the underlying data model design – the core of a model and metadata repository. At first, we give a short overview over these decisions and the dependencies between them (see Figure 6.1). Subsequently, we present each of these decisions in detail. The decision model is distilled from our own experiences, our study of other projects (both open source and commercial), as well as the documented experiences of others.

- *Select Basic Repository Technology*: Usually, one of the first decisions is which basic technology should be used for the repository. Depending on the types and amounts of models or metadata to be stored, either an XML database, a specific file structure, or a standard relational database are alternatives.

- *Select XML to NXD Mapping*: When system architects decide for an XML database, they can select between two basic mapping alternatives, namely an XSD model-based and a text-based approach.

- *Select XML to RDBMS Mapping*: When system architects choose an RDMBS, an important follow-on decision is how to map the XML documents to the database, namely by a domain model mapping or an XSD model mapping.

- *Select XML to File Mapping*: When system architects decide for a file storage solution, they can select between three basic mapping alternatives, namely a XSD model-based, a domain model-based, and a simple text-based approach.

- *Select Repository Type*: Depending on important decision drivers such as searching capabilities and data categorization, system architects can decide for a model repository, a metadata repository, or a model and metadata repository.

- *Select Support for Meta-Model*: When system architects decide for storing models by selecting the model repository and model and metadata repository respectively, they optionally can choose a meta-model that specifies the elements of the stored models.

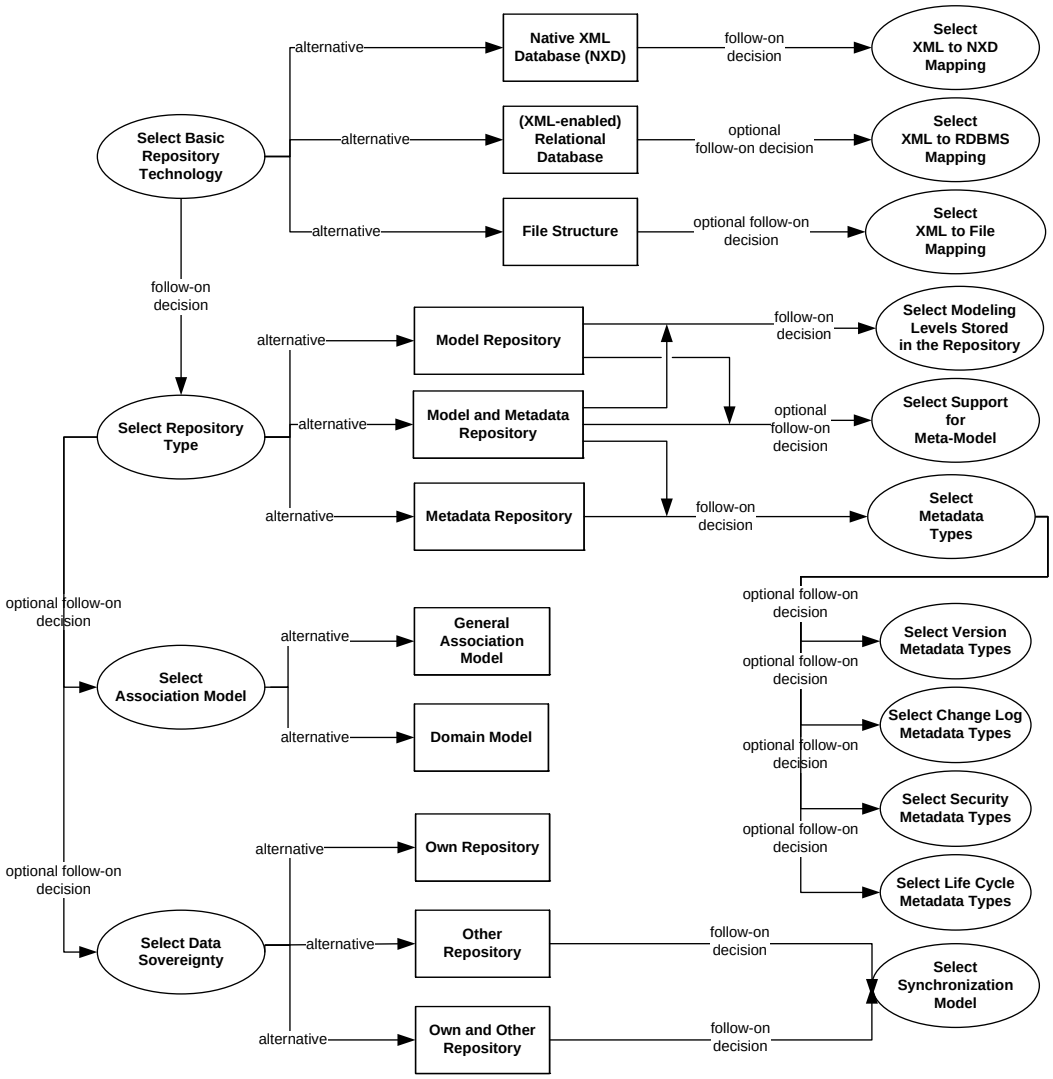


Figure 6.1: Dependencies between Architectural Decisions

- *Select Modeling Levels Stored in the Repository*: System architects have to select the modeling levels such as models, model instances, source code, and runnable code to be stored in a model repository or in a model and metadata repository.

- *Select Metadata Types*: In case a metadata repository or a model and metadata repository is used, system architects can select model-independent metadata such as version information, ownership, affiliations, and security data.

- *Select Version Metadata Types*:

  - *Select Version Modeling Levels*: When selecting version metadata types, firstly, system architects have to decide to which modeling levels to apply versioning.
  - *Select Model Version Granularity*: Afterwards, developers have to make the follow-on decision for choosing an adequate version granularity.
  - *Select Kind Of Model Version Compatibility*: Finally, provided that versioning shall be applied on the model level, they should select a compatibility model for model versions.

- *Select Change Log Metadata Types*: According to the decision of selecting version metadata, system architects have to decide whether to add change log metadata to either the model or model element level.

- *Select Security Metadata Types*: System architects can choose among several security metadata options. Unlike the decisions described before, security metadata does not solely focus on several artifacts, but on mechanisms to secure the whole repository.

- *Select Life Cycle Metadata Types*: System architects can opt for a life cycle manager, that can determine whether a requested action is allowed dependent on the current state.

- *Select Association Model*: This decision deals with whether to model relationships in the domain models themselves or to use a general association model[99].

- *Select Data Sovereignty*: This decision deals with whether the repository shall delegate certain data to owner repositories or other systems.

- *Select Synchronization Model*: When the repository delegates data storing to other repositories/ systems, system architects need to select the kind of synchronization model.

Architectural Decision: Select Basic Repository Technology

A fundamental task of a repository architect is to choose a basic storage technology for the repository. As illustrated in Figure 6.2, there are three basic alternatives for storing artifacts: Native XML databases (NXD), (XML-enabled) RDMBS, and a file system using a specific file structure. RDF triple stores are a popular variant of NXD.

Important decision drivers for this decision are the *amount of data* to be stored in the repository and the expected *performance/throughput* the repository should provide. For developers and administrators it is important to know which *technology know-how* is needed in order to set-up and run the repository technology. One important aspect of the repository technology are the *searching capabilities* provided. When a partner or a customer should be enabled to search for a model or model instance, it is helpful to use a repository based on *standard technology*, as standard interfaces often ease the

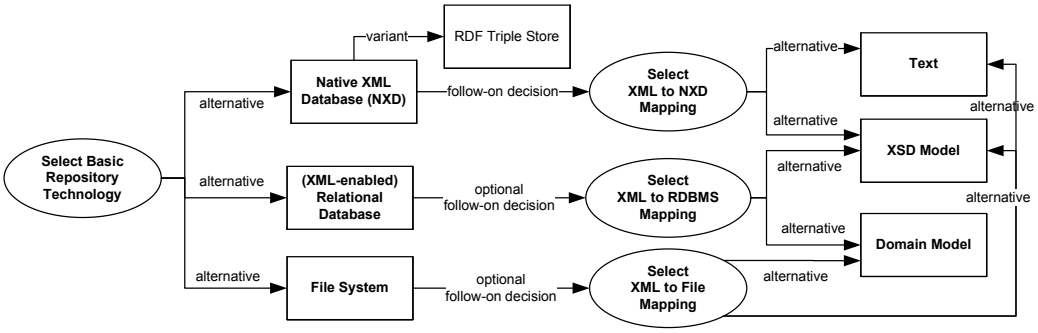


Figure 6.2: Architectural Decision: Select Basic Repository Technology

integration. For standard technologies, often a number of tools and IDE plug-ins exist, which help developers and partners to work with the repository.

There are a number of follow-on decisions related to mapping XML to one of these storage alternatives. Although we mention alternative exchange formats such as objects of a programming language (e.g., as possible in EMF [128]), in the following we focus only on an in-detail description of XML model exchange format mappings, because XML is the common model data exchange format.

"Each of these approaches has its own advantages and limitations" [52]. Particularly with regard to throughput and huge amount of data, a NXF system may work best, because no mapping from XML files to database schemes is required [52]. Furthermore most native XML databases support sophisticated full-text searches [10]. However, due to the document-centric approach, complex queries can have longer response times compared to (XML-enabled) RDBMS systems [10]. One known use of a NXF system is XTC, the XML transformation coordinator for XML document transformation technologies [40].

"Relational databases provide maturity, scalability, portability, and stability" [68], and they are the RDBMS that are probably most widely used today [52]. Known uses of model repositories based on RDMBS are the SWISS–MODEL repository [69] storing "three-dimensional comparative protein structure models" and the BrainML model repository [17] storing neuroscience data.

Alternatively, especially for small amounts of data, system architects could choose a simple file structure as repository storage. For this, one of many known uses is the CellML model repository [78] for storing and exchanging computer-based mathematical models. Of course, when using a file storage, searching a large amount of data, could be rather inefficient in comparison to using either a NXD system or RDBMS. However, for repositories with only small amount of data, this might be the simplest and most appropriate solution. In particular when using proprietary file formats, the repository can be set-up quickly, because no data mapping is required.

Architectural Decision: Select XML to NXD Mapping

Provided that system architects opt for a native XML database, they can decide between two basic storing alternatives (see Figure 6.2). Either the entire XML document can be stored in *text* format or the XML document can be modeled as DOM and mapped to *XSD model* objects such as elements, values, etc.[52]. In the former case the database or file managing component has to manage indices to improve performance on its own. In the latter case, XML documents can be "stored as type-annotated trees on disk pages, indexed with path-specific indexes, and queried with XQuery, SQL/XML"[94].

Whether to use a text-based or an XSD model-based mapping depends on the required *performance* and on the *effort* to establish the system. There are many NXD systems both commercial and open-source. Most XML databases such as DB2[94] support the *XSD model* for mapping XML to corresponding tree structures in NXD[10]. When stakeholders opt for the XSD model-based approach, they can make use of *sophisticated searching capabilities*.

Architectural Decision: Select XML to RDBMS Mapping

Provided an RDMBS database is selected as the basic repository technology and the raw models are provided in XML format, an important follow-on decision is how to perform the mapping between the tree-based XML data model and the rows and columns of a relational data model[10]. System architects can mainly choose between two alternatives: They can either decide to map the *domain model* elements to a database schema or use an *XSD model* approach by mapping standard *XSD model* elements to RDBMS. By using the *domain model* mapping approach, a separate table is generated for each domain model element. In contrast, the *XSD model* mapping approach is characterized by a lesser number of resulting tables and columns, because unlike the *domain model* approach, several XML elements are combined into a single table. Moreover, the resulting RDBMS schema, here, can either be generated from an XSD or from a DTD. An algorithm for mapping XML data to relational data is proposed by Atay et al.[10]. The work of Emadi et.al.[34] compares common DTD-independent methods by performance benchmarking.

Many commercial XML-enabled database systems such as SQL Server and Oracle support both the *XSD model* and the *domain model* mapping. In the latter approach the existing data model is extended with an additional XML data type to query and store XML data[10].

Basic decision drivers are both *performance* and the *effort* to accomplish the mapping. In case neither an XSD nor a DTD exists, system architects should decide to use the *XSD model* mapping approach. Additionally, this approach can "reduce the number of join operations incurred while querying the data"[52]. Florescu's and Kossmann's work[38] shows that even simple approaches provide good performance. Thus, in most cases, we would clearly recommend to use the *XSD model* mapping approach, especially if performance is the most important decision driver. However, a lower effort to establish the mapping can be achieved with the *domain model* approach. In certain *development environments*, e.g. in object-oriented systems, system architects may prefer the *domain*

model mapping alternative, because it is less abstract and reflects the object-oriented view of the developers.

Architectural Decision: Select XML to File Mapping

In case system architects decide to use an appropriate file system structure and the models are stored as XML documents, they can select among three basic storing alternatives (see Figure 6.2). The file itself can contain the entire XML document as *text*, the XML document can be separated according to the *XSD* model, or the document can be split into several files according to its *domain model*.

The advantages and disadvantages for using the XSD model-based or the domain model-based approach were already discussed above. The obvious advantages of the text-based alternative are simplicity and the low effort to establish the system. Thus which alternative to use depends on the required *performance* and on the *effort* to establish the file system storage.

Architectural Decision: Select Repository Type

Depending on the repository's functional requirements, models, model instances, and/or metadata must be stored in a model and/or metadata repository. As already defined in Section 6.2, we can distinguish three alternative repository types: *model repository*, *metadata repository*, and *model and metadata repository*. Figure 6.1 depicts these alternatives. Most repositories use metadata to describe general characteristics such as version information, user information, and security data. In contrast to metadata, models contain domain-specific elements. Some metadata, such as version information, is linked to specific models as add-on data, other metadata, such as user authorization data, can be considered as general repository data that is not linked to specific model data. In Section 6.3 we focus on selecting adequate metadata types.

The decision drivers for storing models in the repository are mainly *functional requirements*. Examples are: An important decision for system architects is whether the MDD paradigm[139] should be supported using the repository architecture. When using MDD, the source code is generated from the underlying models and these models must be accessible from the repository. In Section 3.2 we stated that a repository should provide *query mechanisms* to search for repository artifacts according to certain search criteria. These query mechanisms are based on categorized data such as domain specific model data and repository metadata. When system architects want to store non-model artifacts, in order to provide appropriate searching mechanisms, they should at least provide these artifacts with some add-on metadata. Accordingly, in case solely non-model artifacts are stored in the repository and provided with add-on metadata, system architects decide in favor of a *metadata repository*.

System architects will choose a *model repository*, if they intend to store models in the repository and do not require additional metadata, because the domain models possibly contain part of this information. Moreover, adding special-purpose metadata such as

ownership and affiliation information to repositories in small companies may not be necessary.

When more sophisticated queries about the repository artifacts are required, system architects should consider storing categorized model data and thus select the *model and metadata repository* alternative. A known use of a *model and metadata repository* is the data access service (DAS) repository that we developed during our studies. In our case study (see Section 6.4) we give more details about the DAS repository when applying it to our reusable architecture decision model.

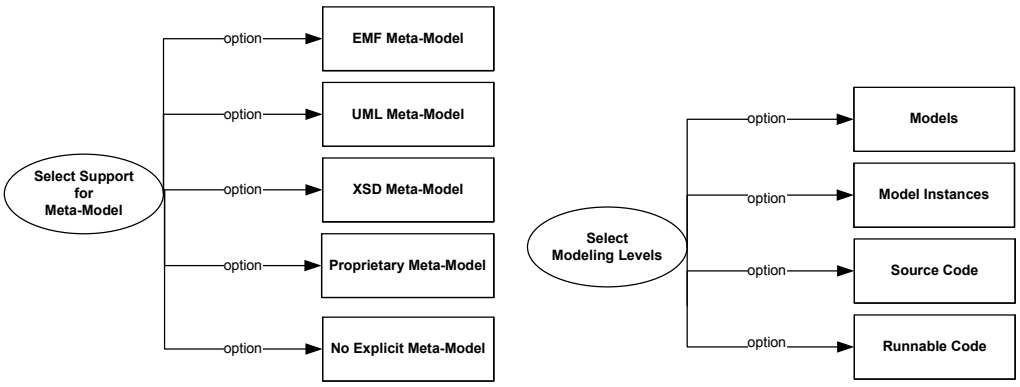
Once this decision has been made and if we have decided for one of the alternatives that include metadata, we would need to make a follow-on design decision: Selecting the types of metadata that are represented in the repository. Accordingly, if we have decided for one of the alternatives that include modeling data, we would need to make one or two follow-on decisions: An optional follow-on decision is to select support for meta-models, and an mandatory decision is to choose the modeling levels stored in the repository.

Architectural Decision: Select Support for Meta-Model

Provided that system architects decide for a *model repository* or a *model and metadata repository*, they can select a meta-model for the domain models to be stored. A meta-model describes the underlying model and thus is the basis for model validation by tools. Moreover, compared to purely textual specifications, meta-models enable a much more compact and clear overview of the model[33]. In addition, meta-models such as UML and EMF[128] can support visualizing models and thus ease model readability and understandability. It has also been demonstrated that a meta-model can be used to compare heterogeneous models. In the literature there are various approaches addressing the problem of heterogeneous model integration[23,93].

Decision drivers are both the *functional* and *technical requirements*. Firstly, system architects may use an explicit meta-model, if they wish to benefit from one or more of the properties described above. Secondly, technology reasons such as using MDD[139] can be a determining factor for using a meta-model. In case of MDD[139], system architects profit from tool support. For instance, they can use a meta-model-based generator, such as EMF's[128] ant task *emf.Ecore2Java*, to generate source code from the models specified by a corresponding meta-model such as EMF[128]. If system architects do not want to profit from these functional and technical features, they can make use of a simple, but much less flexible approach: To support no explicit meta-model. That means, to hard-code the meta-model information and thus specifying a model without an underlying meta-model.

In addition to that option, in Figure 6.3(a) we illustrate several meta-model options among which system architects can select: EMF[128], UML, XSD, and a proprietary domain meta-model (see Figure 6.3(a)). They should choose a proprietary domain meta-model, if standard meta-models such as UML and EMF[128] did not fulfill the requirements.



(a) Select Support for Meta-Model (b) Select Modeling Levels

Figure 6.3: Architectural Decisions: Select Support for Meta-Models and Modeling Levels

A known use of using EMF[128] meta-models, is our data access service (DAS) repository that we developed during our studies. A known use of a model repository that loads UML2 models into EMF is the AndroMDA's EMF UML2 repository[6]. In contrast, the BrainML model repository[17] has a proprietary BrainML meta-model, that conforms to the standard XML schema.

Architectural Decision: Select Modeling Levels Stored in the Repository

Provided that system architects choose a *model repository* or a *model and metadata repository*, an important follow-on decision is to select the modeling levels stored in the repository: Models, model instances, source code, runnable (byte) code, or all of them.

Figure 6.3(b) depicts this architectural decision and its four modeling layer options. In the following we specify important decision drivers for each of these layers.

At first, system architects should face the question whether to store models or not. In this context an important decision driver is *automatic validation* of new models and model instances. When storing models in addition to model instances, the model instances can be validated using their models. In order to accomplish this validation, the model instances have to be linked with their specific models. Accordingly, if an automatic syntax-check fails, the publishing request can be rejected by the repository. Furthermore, in an extended version, the repository could try to *automatically adapt* existing model instances when the underlying model changes. When system architects do not want to profit from the advantages of automatic syntax checking and automatic adaption of source code, they can ignore the model layer in favor of saving *storage space* and *effort*.

The next decision is whether system architects should store model instances in the repository. This decision is closely related to the required *search capabilities*. Besides the

desired search capabilities, another decision driver is whether to support MDD or not. In case MDD is supported, model instances rather than source code are stored in the repository because the generator can use transformations to generate the source code. In some cases, this means that the transformations for the generator should also be placed in the repository. However, even for non-model-driven projects, we recommend to store model instances, especially because to support querying artifacts.

There is also the option to store the model instances but not the models. An example of a known use that stores model instances, but no models, is the Eclipse CDO Project [32].

Whether the repository should provide source code, depends both on the *technical requirements*, such as using MDD [139], and on the *development environment* and *platform* of repository users. When MDD is used, commonly *technology- and platform-independent* model instances are stored in the repository. Accordingly, on the client side, repository users can generate source code from these model instances according to their specific platform- and technology requirements. Thus, if more than one technology or platform is supported, source code shall not be stored in the repository, but generated by the repository users. Otherwise, if no technology- and platform-dependent source code generators are required, system architects can decide to store the source code in the repository. In this case, generated source code can also be stored in the repository. Alternatively, the source code can be stored in an external repository. In this case, a reference to this external repository can be specified e.g. in the model instances or in appropriate metadata (for more information about selecting metadata types please refer to Section 6.3).

The next decision system architects should make is whether the repository should supply runnable byte code and how. In the following we present three alternatives: The first alternative proposes to build the source code on the client side. This alternative primarily depends on the users' *source code build environment* that has to fulfill the *technical requirements* to build the source code. The second alternative discusses storing the byte code in the repository itself. A disadvantage of this alternative are the associated *storage costs*. An advantage is that *building the source code* on the client side is not necessary. The third alternative only provides metadata about where and how to locate a runnable software component. From the users' point of view, this alternative is probably the simplest one. However, for technical reasons, such as performance issues, system architects could reject this alternative and decide in favor of storing or building the byte code.

Architectural Decision: Select Metadata Types

Common repositories include metadata to provide additional, model-independent information of repository artifacts. Figure 6.4 shows a few options: Metadata can include versioning information, change log data, ownership and/or affiliation information, security data such as information on role-based access control and identity management, location information, life cycle data, and data for internationalization features (see Section 6.3). In the following we give a detailed overview of each of these metadata options commonly used in repositories. System architects can use this checklist to decide whether

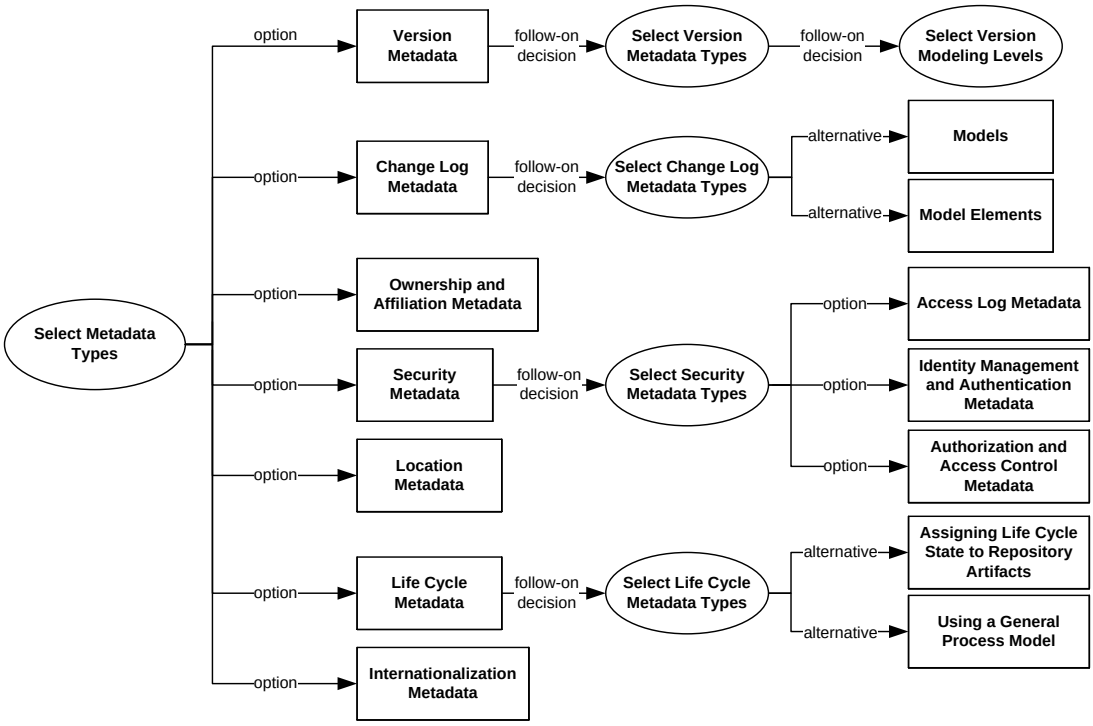


Figure 6.4: Architectural Decision: Select Metadata Types

to apply a certain metadata type or not. We have developed this checklist by studying common repositories to the best of our knowledge. However, due to the diversity of possible metadata types, the list is not exhaustive. After illustrating the checklist, in the proceeding sections we particularly focus on the follow-on decisions as well as resulting options and alternatives depicted in Figure 6.4.

Version Information Metadata System architects have to decide whether to add version metadata or not. In the simple case, system architects can opt for using no versioning. For this purpose, they solely need to provide the *most recent version* of repository artifacts. Otherwise, if the repository supports version management, they shall make the follow-on decision of selecting version metadata types.

Change Log Metadata Change log metadata can include information about which user has inserted or updated a certain repository artifact. The decision whether to add change log data is based on the previous decision of adding *version information metadata*. Thus, system architects can not opt for providing change log metadata, not until they decide in favor of using version metadata. Below, we present the follow-on decision of selecting different change log metadata types.

Ownership and Affiliation Metadata System architects can decide to tag repository artifacts with ownership and affiliation metadata. This information can contain name, contact details, and affiliation information of repository artifact owners. By using this metadata, system architects can enhance *reuse* of stored artifacts such as models, model instances and source code. Adding this metadata and thus being able to search for specific artifacts, is especially essential in *large and medium-sized companies*. If, however, stored repository artifacts are intended to be solely used by a *small team* of developers anyway, system architects can determine to omit this type of metadata.

Security Metadata According to their security requirements, system architects can choose one or more types of security metadata (see the ebXML registry services and protocols specification[99]). Please note that, unlike other types of metadata, security metadata does not solely focus on several artifacts, but on mechanisms to secure the whole repository. Below, we present some basic security options system architects can install.

Location Metadata Another type of metadata, system architects can choose, is location metadata. As already mentioned before, source code and runnable code can be linked to models and model instances stored in other repositories. The decision drivers for deciding whether source code and runnable (byte) code should be stored in the repository itself or in an external repository are the same as those described in Section 6.3. Besides source code and runnable code, location metadata can be important, e.g., for linking model instances or source code to specific documentation on document servers. In order to *save storage cost* and *maintenance efforts*, we recommend to decide in favor of referring to existing documentation instead of storing this information redundantly.

Life Cycle Metadata A repository incorporating life cycle metadata manages all life cycle actions such as inserting, updating, deleting, and deprecating repository artifacts. Besides these basic actions, the life cycle manager can oversee further actions such as validating model instances, and finally publishing changes to repository users. Depending on the current life cycle state, the life cycle manager determines whether the requested action is allowed and consequently performs or rejects the action. Below, we present the follow-on decision of selecting suitable life cycle metadata types.

Internationalization Metadata Internationalization metadata can be used for storing location-specific settings, such as different languages and coding sets. In the EBXML standard[99] internationalization metadata is defined as attributes, that may be localized into multiple native languages. System architects will choose internationalization metadata, if e.g. *international project members* shall access the repository or *different coding sets* shall be supported.

Architectural Decision: Select Version Metadata Types

In order to select version metadata types, system architects, firstly, have to make the primary decision at which modeling levels to apply version metadata types. Secondly, they have to make two important follow-on decisions, namely to select version granularity and to select the kind of model version compatibility.

Architectural Decision: Select Version Modeling Levels

When selecting version metadata, system architects have to select the modeling levels at which to apply version information. At least, they have to opt for the *model* and/ or *model instance* level (see Figure 6.5).

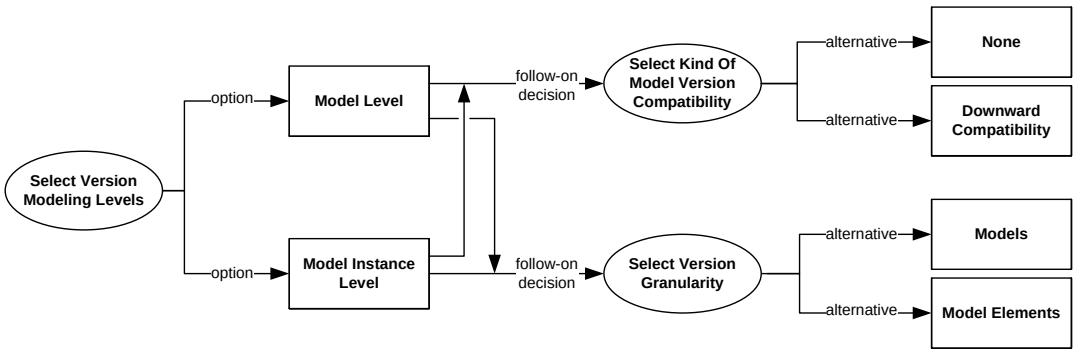


Figure 6.5: Architectural Decision: Select Version Modeling Levels

An important decision driver is the *high complexity* arising when storing model and model instance versions. When system architects decide for managing versions in the repository, they have to face consistency and compatibility problems, because model instances of different versions can comply to different model versions. Another important decision driver is the *maintainability effort*. If system architects decide against versioning models, whenever a model changes, the model instances will have to be regularly upgraded to comply with this new and only model version. Therefore, if the diversity of model instances in the repository is considerably low, system architects shall opt for solely managing model instance versions. In this case, upgrading the model instances can be established with considerably low effort. Otherwise, if various different model instances are managed by the repository, it will be the best solution to manage versions for models and model instances.

Architectural Decision: Select Kind Of Model Version Compatibility

In contrast to managing model instances of the same model version, managing model instances belonging to different model versions comes along with consistency problems. Thus, system architects have to decide for the kind of model version compatibility.

Firstly, system architects can opt for using *no version compatibility* for model versions. In this case, new model versions can be created with no restricts regarding to compatibility issues for previous model versions. Further, they can decide on the model *downward compatibility* alternative. In this case, each new model version has to be downward compatible with each of its previous versions. When system architects decide for being downward compatible with previous model versions, new model versions may contain new model elements, but must not contain renamed or deleted elements.

The first decision driver, system architects have to consider, is the *increasing complexity for querying model instances*. If a model element of a new model version has been deleted or renamed, the querying of model instances could become very complex because different model versions would have to be managed. The second decision driver is the *quality of the search results*. If a model element of a new model version has been renamed, search results would be dissatisfying, because not all expecting model elements would be part of the result set. A third decision driver is the *flexibility to define new models*. If a high flexibility is desired, system architects shall opt for none compatibility. Downward compatibility comes along with limited flexibility to create new models. For instance, if a certain element is not necessary anymore, it must not be deleted. Otherwise the new model version would not have been downward compatible with the previous versions.

Architectural Decision: Select Version Granularity

When storing models and model instances, system architects can decide on either adding version information metadata to the whole model/ model instance or to each model/ model instance element. The CellML model repository[78] is a known use of a repository, that stores version information at the model/ model instance level. If a CellML model/ model instance is modified, the new updated version(s) will be added to the repository[78]. The BrainML model repository[17] is another known use that adds version information metadata at the model/ model instance level. Version numbers start at one and are incremented whenever an augmented or modified version of the model/ model instance is submitted. Earlier versions remain available in the repository and can be referenced by their version number.

Standards such as UDDI[25], EbXML[99], and the Content Repository API for Java Technology of Java Specification Request (JSR)[96] support adding version information to model elements and model instance elements, respectively. JCR consists of one or more workspaces that each consists of a tree of items representing either nodes or properties. A content repository[96] workspace that supports versioning may contain both versionable and non-versionable nodes. A known use open-source implementation variant of a Java content repository is eXo JCR[36]. According to the JCR , eXo JCR supports separate versioning of repository artifacts such as model elements.

The decision, which of the alternatives to select, depends on the type of update-strategy in case of changes. If selective updates are desirable, we will recommend using versioning for *model elements* and *model instance elements*, respectively. If artifacts

such as models are updated as a whole, the alternative of versioning *models* and *model instances* rather than *model elements* and *model instance elements* shall be chosen.

Architectural Decision: Select Change Log Metadata Types

The decision whether to set-up versioning on the model or model element level is closely related to the question how fine-grained changes need to be traced and monitored. When choosing the alternative to version *model elements*, a specific event log of changes for each model element is stored. An alternative is the versioning of *models* where an event log of changes is only available at the model level.

If system architects want to provide change log data at the model element level, the corresponding change log information at the model level can be a view of all related model element change log data. Moreover, when system architects only need change logging at the model level, they save *effort* compared to storing logs at the model element level. However, if system architects already decided in favor of *versioning*, the change log information should be set-up at the same model and model element level, respectively, as selected for the previous version management decision.

Architectural Decision: Select Security Metadata Types

Provided that system architects settled for storing security metadata, they can decide in favor of one or more of the following options.

The first option is to provide *access log metadata*. Hereby, the repository keeps a journal of all significant actions performed by repository requesters on repository resources. Another option is to establish *identity management and authentication*. Choosing this option means, the repository itself manages the identity and credentials associated with authorized users and services. Finally, system architects can enable authorized users to perform specific actions or to access specific resources by establishing the *authorization and access control* option. The repository provides a mechanism to protect its resources from unauthorized access. In this context, system architects can augment a role-based access control solution with well-defined authorizations for each role.

Architectural Decision: Select Life Cycle Metadata Types

In this decision, system architects have two basic alternatives: They can either *assign a life cycle state to each repository artifact* or implement a *general process model* containing flows of activities. In the latter case, a process engine is needed to drive the execution of activities[14]. When deciding for the first alternative, the *complexity* of the life cycle grows much more than proportional by the number of life cycle states. However, if system architects intend to use only *basic life cycle actions* such as insert, update, and delete, this alternative will be a very effective one.

A known use implementation incorporating life cycle metadata is the ebXML registry reference implementation project[43]. The ebXMLRR project aims at delivering a functionally complete reference implementation for the OASIS ebXML specification[99].

According to the ebXML standard, each *RegistryObject* instance must have a life cycle status indicator that is assigned by the registry. In contrast, the alternative of using a general process model should be used, when there are potentially new actions that will be developed in future. Accordingly, if system architects attach a great value on *life cycle scalability*, they shall decide in favor of a general life cycle model.

Architectural Decision: Select Association Model

Modeling associations among models and model instances is a commonly addressed problem today. However, a current problem in process-driven SOAs is to retrieve the relationships between different components, such as which service operations can be invoked from which process activities and which services access which data. Furthermore, components that do not depend on any component can be seen as obsolete, and thus can be deleted [83]. Another benefit from modeling dependencies between different components is to visualize these dependencies to better support understandability of the models. For this purpose, graphical tools can be designed, because the tools are what give value to a repository [13].

As shown in Figure 6.1, there are two basic alternatives among which system architects can choose: As described in our fundamental work of VbDMF [83], general models can specify associations between certain special-purpose models. In the example, our view-based data modeling framework (VbDMF) describes the associations between processes, services, and underlying persistent data access. If domain models do not specify associations between the artifacts, the repository shall handle these associations by defining a general association model as specified in the EbXML standard [99]. EbXML's association information model defines classes that enable artifact instances to be associated with each other.

Architectural Decision: Select Data Sovereignty

In this decision, for each kind of repository artifact, system architects have to decide whether to *only store the data in the repository itself*, or to *store the data in the owner repository without storing the data in the repository itself*, or *to redundantly store the data in the repository and in the owner repository.* System architects can select for storing the data solely in the repository itself, if the *data does not logically belong to another systems or repositories.* This is likely to be the case for *models*, because they are usually only repository-internally-used, basically in order to specify the model instances. Otherwise, if *another repository owns the data*, system architects can opt to both store the data in the owner repository and store the data in the repository itself, or solely store the data in the owner repository. When the owner repository does not support *structured elements*, system architects can opt to store the data redundantly in both repositories, because without structured elements, no sophisticated querying is possible. Another decision driver, for whether to store data redundantly in both repositories or solely in the owner repository, is the *availability* of the data in case the owner repository is temporarily not available.

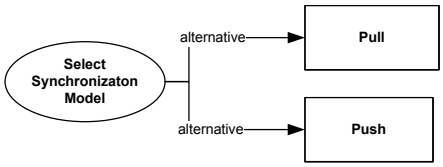


Figure 6.6: Architectural Decision: Select Synchronization Model

Architectural Decision: Select Synchronization Model

When storing data redundantly to both the repository and its related repositories such as the service repositories, system architects have to deal with synchronization problems. In order to synchronize data between the repository and its owner repositories, the data has to be replicated from the repository to the owner repositories and vice versa.

As shown in Figure 6.6, in order to synchronize data, system architects can choose between two alternatives, namely the *push-based* and *pull-based* synchronization model. When using a pull-based synchronization model, the data artifacts are sent from the repository to the other owner repositories or vice versa without delegating the requests[2]. For this, system architects have to *integrate active monitoring mechanisms* to transfer changes e.g. from other repositories such as service repositories, in example UDDI[25], to the repository[30]. By these active monitoring mechanisms, the latest information of UDDI can be found transparently and conveniently[30]. In order to implement a pull-based synchronization model, system architects have to carefully balance the number of data replications. On one hand, frequent data replications have to be done to synchronize the repositories. However, these will be inefficient, if the repository data seldom change during a certain interval. On the other hand, without frequent crawling, the repository content may become inconsistent with the other repository[1].

When deciding for the push-based model, the repository acts as a Push Model Data provider to directly push the requests to other repositories such as to service repository side.

An important decision driver is whether the *owner repositories already implement a synchronization model*. If this is the case, system architects can exploit this model in order to replicate and transform the data from the repository to the owner repositories. However, often, repositories, in example service repositories such as UDDI[25], implement no synchronization model, neither the pull-based nor the push-based synchronization model. Thus, in this case, in order to synchronize data, system architects have to opt for the pull-based model to synchronize data from the owner repository to the repository, and likewise, they decide for the push-based model to synchronize data from the repository to the owner repository.

Furthermore, an important decision driver for using the *push-based model* is whether the other repositories *provide services* that enable requesters to store artifacts in the repository. A further decision driver for using the push-based model is the *availability* of these provided services. When the services are not available twenty-four hours a

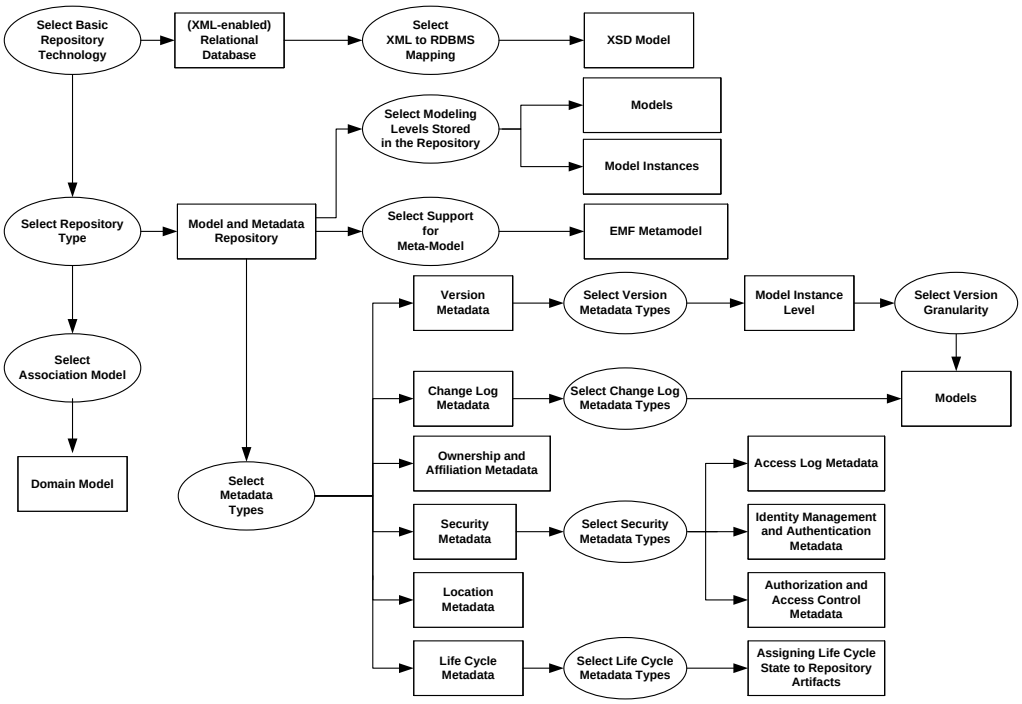


Figure 6.7: Case Study: Selected Decisions for a Data Access Service (DAS) Repository

day, seven days a week, system architects can decide for a pull-based synchronization, provided that the owner repositories implement such a pull-based synchronization model.

6.4 Case Study

In this case study we illustrate major design decisions, that we made when setting-up our own data access service (DAS) repository. During the design process of the DAS repository we were faced with several fundamental architectural decisions. Here, we reflect the decisions made to set-up our DAS repository by walking through the reusable architectural decision model depicted in Figure 6.7 of Section 6.3.

Before we walk step by step through our reusable architecture decision model, we would like to shortly motivate the use of a Data Access Service (DAS) model and metadata repository: Developers typically store DAS in local file systems and concurrent versioning systems, such as CVS or SVN. However, especially as the number of DAS grows, finding a particular DAS stored in a concurrent versioning system, in order to reuse the DAS, can become rather time-consuming. Thus, developers need more sophisticated query mechanisms to quickly locate existing DAS operations in order to increase DAS reuse. For example, the DAS repository supports queries for retrieving desired DAS by diverse search criteria, such as finding all DAS accessing a particular database,

all DAS operations inserting data into a particular table, or all DAS operations updating a certain column of a table. Moreover, with the DAS repository, DAS developers are able to query ownership information about a certain DAS and thus look for all DAS registered by a certain user or department.

1. *Select Basic Repository Technology*: When setting-up our DAS repository, we decided in favor of the *(XML-enabled) relational database* alternative. Our main decision driver was that RDBMS are very common and hence, we can benefit from tool support. We decided against using a *NXD* system, because our DAS repository models have many associations between them, and thus many joins are necessary when querying DAS data. Accordingly, they are the joins in *NXD* storages, that can have longer response times compared to RDBMS. As searching a large number of DAS could be rather inefficient, for us, a *file system* storage was out of question.

2. *Select XML to RDBMS Mapping*: We decided in favor of the *domain mapping model*, because, in our object-oriented programming environment, it is less abstract and better reflects the relationships between the objects.

3. *Select Repository Type*: Our DAS repository was designed to primarily manage models and model instances, but also to be defined metadata. As a consequence we opted for the *model and metadata repository* alternative.

4. *Select Support for Meta-Model*: We chose *EMF*[128] as an explicit meta-model so that we can specify our VbMF/ VbDMF models. Thus, we can benefit from existing tool support such as the Eclipse Model To Text (M2T) project's Xpand language[129] to generate source code from the models e.g. we can generate plain old Java objects (POJOs)[65] describing a WSDL[140].

5. *Select Modeling Levels Stored in the Repository*: We decided to store both the view model instances and the view models for the following reasons: We store the view model instances because they contain the basic elements that can be, in particular, applied to generate the runnable DAS. Our generated DAS source is dependent on the specific object relational mapping (ORM) technology such as HIBERNATE[55] or IBATIS[57]. For this purpose, our DAS repository stores technology- and platform-dependent *model instances*, that are used for source code generation on the server side. We store the view models in the repository to enable important development aspects such as validating the view model instances. Another requirement was to automatically validate checked-in model instances. In order meet this requirement, we settled for storing *models* in addition to model instances.

   Our repository centrally generates the source code from the model instances and builds the runnable code by compiling the generated source code. Thus, our repository neither stores source code nor runnable code.

6. *Select Metadata Types*: As we opted for a model and metadata repository, we also added metadata to our repository. Afterwards, we focused on those decisions that are not covered by follow-on decisions: We settled for adding *ownership and affiliation* metadata in order to being able to trace which user registers and/ or publishes which artifact to the repository. Up-to-now, the repository does not link to documentation or to source code stored in other repositories. Thus, the repository does not manage *location metadata*. As our DAS repository still is a prototype solution, at the moment, *internationalization* metadata is not provided.

7. *Select Version Metadata Types*: We opted for versioning support. Thus the following decisions had to be made:

   - *Select Version Modeling Levels* As the diversity of model instances in the DAS repository is considerably low, we decided to solely manage model instance versions. Accordingly, upgrading the DAS model instances to the newest model version can be established with considerably low effort. Thus, we had no need to decide the kind of model version compatibility.
   - *Select Version Granularity* We wanted to save extra efforts, thus we did not opt to version model elements. Thus, we decided in favor of adding version information metadata to *whole models and model instances*.

8. *Select Change Log Metadata Types*: As this decision is based on the decision of selecting version metadata types, we opted for adding *change log metadata* to *models and model instances* in contrast to *model and model instance elements*.

9. *Select Security Metadata Types*: As we intend to provide our repository to industry, we added basic *security* metadata for all three security options illustrated before, namely *access log metadata*, *identity management and authentication metadata* and *authorization and access control metadata*.

10. *Select Life Cycle Metadata Types*: Our repository incorporates a basic *life cycle manager* , that manages basic actions such as insert, update, delete and validate. As we required both a simple solution and the life cycle manager not necessarily to be scalable related to new actions and states, we opted for *assigning a life cycle state to repository artifacts*. We have decided against using a *general process model*, because this solution seems a bit over-sized for our prototype repository solution.

11. *Select Association Model*: Our DAS models specify relationships between each other. Thus, we use our own *domain models* to specify associations between DAS model instances instead of using a *general association model*.

12. *Select Data Sovereignty*: Basic VbDMF models such as the Collaboration View model and the Information View model specify a web service description language (WSDL). Thus, these view models have to be owned by a service repository. Unfortunately, the service repository specification such as UDDI[25] does not support structured WSDL elements. For this reason we decided to store the view model

instances redundantly both in the owner repository and in the DAS repository instead of storing the data solely in the service repository. We do not need to store the view models redundantly, because they are only DAS repository internally used.

13. *Select Synchronization Model*: Our DAS repository uses the push-based synchronization model to directly push publication requests from the repository to service repositories. Services repositories such as UDDI[25] do not implement any synchronization model, but provide services to enable requesters to store artifacts in the repositories.

6.5 Summary

In this chapter we introduced a reusable architecture decision model (RADM) for setting-up model and metadata repositories. These decisions in particular focus on database design for model and metadata repositories. We provided a decision basis for fundamental choices such as selecting a basic repository technology, choosing appropriate repository metadata, and selecting at which modeling levels information shall be stored in the repository. Our experiences result from developing our own model repositories, from researching on other works, discussions with other people involved in repository projects, and applying our RADM to a case study.

CHAPTER 7

View-Based Model-Driven Architecture for Enhancing Maintainability of Data Access Services

In this chapter we bridge the gap between the DAS and their implementation by presenting a view-based, model-driven data access architecture (VMDA) managing models of the DAS, DAOs and database queries in a queryable manner. Our models support tailored views for different stakeholders and are scalable with all types of DAS implementations. Our view-based and model driven architecture approach can enhance software development productivity and maintainability by improving DAS documentation. Moreover, our VMDA opens a wide range of applications such as evaluating DAS usage for DAS performance optimization. Furthermore, we provide tool support and illustrate the applicability of our VMDA in a large-scale case study. Finally, we quantitatively prove that our approach performs with acceptable response times.

This chapter is organized as follows: First, Section 7.1 we motivate our approach. Next, in Section 7.2 we present our view-based model-driven data access architecture (VMDA). Section 7.3 looks deeper into the DAS repository by describing the underlying DAS repository services and the data model. The following Section 7.4 presents our prototype tooling and hence describes our VMDA from the user's point of view. In Section 7.5, we illustrate the applicability of our approach by a real-life case study in the area of geographic information systems (GIS) in the context of web feature services (WFS). Section 7.6 underlays our approach contributions with quantitative evidences. Finally, Section 7.7 sums up.

7.1 Motivation

In service-oriented architectures, service providers publish their services to service repositories in order to enable service consumers to dynamically locate and bind the services. In a process-driven SOA, the process activities can query a service repository in order to find suitable services for dynamic invocation (see Figure 7.1). As a result, the service repository returns the required services running on a service provider.

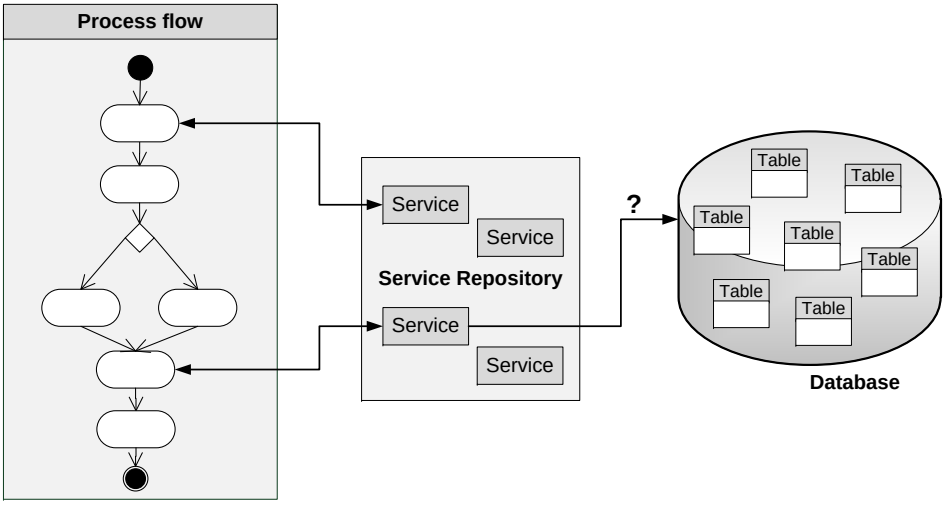


Figure 7.1: Missing Link between Services and Data

However, unfortunately, up-to-now there is no adequate managing solution for organizations in order to locate data access services (DAS) in process-driven SOAs. This is because there still is a gap between the service repository and the physical data storages. In smaller environments, development techniques like naming conventions and documentations may be also be useful to manage DAS in process-driven SOAs. However, in larger-scale environments, when the number of software components grows, more sophisticated methods to manage traceability and maintainability are necessary. Therefore, in this chapter we provide an architecture solution to enhance both maintainability and reuse of DAS in process-driven SOAs.

7.2 Architecture Overview

In this section we present the big picture of our view-based model-driven data access architecture (VMDA). Our VMDA unifies the following four contributions:

1. Using DAS to be independent of the underlying data sources
2. Specifying DAS models/ model instances making use of the advantages of MDD

3. Applying VbDMF/ VbMF to separate the DAS models/ model instances into different view models/ views

4. Establishing a DAS repository to manage the DAS view models and views

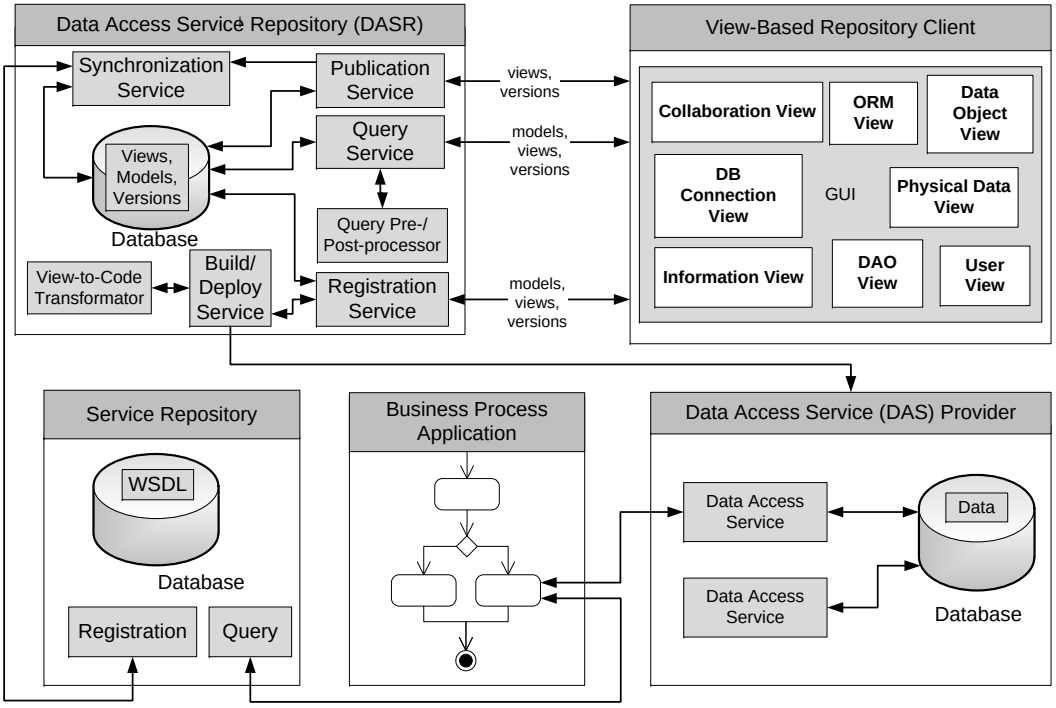


Figure 7.2: View-Based Model-Driven Data Access Architecture (VMDA)

There are many solutions of use based on model repositories to efficiently manage structured elements[69],[17]. In our work, we also take advantage of such a *model and metadata repository* to *manage* both the *DAS/DAO view models specifying* the *DAS/ DAO view model instances* and the DAS/ DAO view model instances describing the specific *DAS* and *DAOs*. Our **data access service (DAS) repository** is the main part of our VMDA depicted in Figure 7.2. The DAS repository is used to manage the view model instances and view models of the DAS, DAOs, the underling data storage schemes, and the relationships between them.

In order to manage these view models and view model instances properly, the DAS repository provides a **query service** to discover the DAS and the underlying DAOs by different search criteria. Via the query service both view models and view model instances can be retrieved. Our query service uses a **query pre-/ post-processor** to perform query transformations to translate proprietary query languages into valid database queries. One of many possible implementations to translate proprietary, platform-independent query languages into platform-dependent languages can be found in[15]. In

Section 7.4 we define our own platform-independent proprietary query language that is used by our prototype implementation to query view models and views by different search criteria.

The **registration service** stores the view models and view model instances in the DAS repository. After registering a view model or view model instance, only a limited group of persons is permitted to query the registered information. In order to enable this group to test the registered views, the registration service can invoke the build/ deploy service in order to make the DAS views available on a certain **data access service (DAS) provider**. In order to give another few developmental aspects, the **build/ deploy service** uses a **view-to-code transformator** in order to generate source code from the defined view model instances. For source code generation, we use the Xpand language of the Eclipse Model To Text (M2T) project[129]. After source code generation, the DAS can be automatically built and deployed on a certain **DAS provider**. The DAS provider information can be either set by the repository client or by the DAS repository's build/ deploy service.

After successfully testing a DAS/ DAO, the views can be published to selected persons, teams, departments, or companies. In order to accomplish this, DAS repository clients have to publish the DAS views via the **publication service**. Once a DAS is published, it can be queried by extended user groups. In addition, once deployed, users of other repositories such as service repositories can be informed about the deployed services. Besides views, users can publish view models in order to provide them to other users and groups. The publication service invokes the **synchronization service** to publish the DAS views to other repositories. This will be the case, i.e. if a DAS endpoint of a service model, stored redundantly in both the DAS repository and in a **service repository**, changes. If so, there will be a need to replicate the new DAS endpoint from the service repository to the DAS repository or inversely. The synchronization service replicates view model instances, but no view models. The reason for this is that the view models specify the view model instances and are thus dedicated only to the DAS repository. In addition, besides publishing the views to the service repository, we could also replicate views to other repositories. In example, storage schema relevant views such as the Physical Data View could be synchronized with a schema repository[16].

The **business process application** can invoke the deployed DAS running on a specific DAS provider endpoint. Moreover, according to a process-driven SOA, a business process can dynamically query suitable services from the service repository and invoke these deployed services on a certain DAS provider. By the way, in this thesis we do not focus on dynamic invocation of services, however, in order to describe our architecture concept we keep the whole SOA in view. We use a **view-based repository client** based on VbMF and VbDMF, in order to enable users to comfortably specify the view models/ views and to access the repository services. The view-based repository client exploits the concept of separation of concerns and hereby improves maintainability of the data access in process-driven SOAs[120]. We present our prototyped view-based repository client implementation in Section 7.4.

In this section we have given a basic overview of our VMDA. In the next section we describe the central DAS repository in more detail.

7.3 The Data Access Service (DAS) Repository

As the DAS repository is the central component of our VMDA, firstly, we present the services for querying, registering, publishing and synchronizing the DAS repository artifacts in more detail. Secondly, we focus on the entities of the repository's view-based data model.

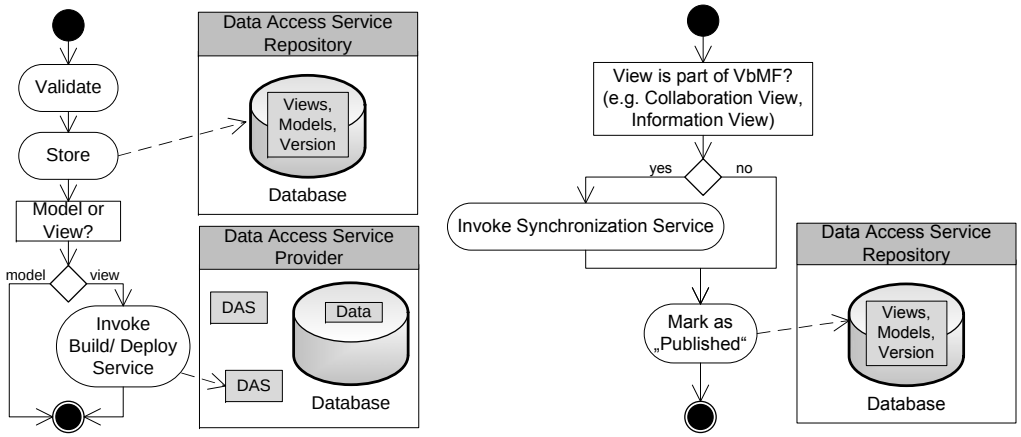
Services

In the following we explain the DAS repository services in more detail. As illustrated in the case study Section 6.4 of the precedent RADM chapter 6, all services of the DAS repository support both view models and view model instances.

Query service The query service is the most powerful part of the DAS repository's service interface. If a column of a database table has to be modified, how to find out which DAS will have to be redeployed? Typically, the modified column is part of the object-relational mapping that in turn is encapsulated by a DAO invoked by a DAS. Thus, in order to best-possibly connect the DAS, DAOs, the object-relational mappings, and the columns and table of data storage schemes, a structured search is necessary. The basic idea of the query service is to retrieve DAS by different search criteria. e.g. by the query service we can query DAS not only by the DAS name, but by implementation, data storage schema and metadata artifacts. Examples of these implementation artifacts are the member variables of data objects that can be mapped to columns of database tables by object-relational mapping (ORM) frameworks. Another example of implementation artifacts are DAO operations that encapsulate the data access queries in object-oriented languages. Examples of data storage schema artifacts include data storage components such as columns, tables, and databases. Moreover, metadata artifacts such as affiliation and version information can be used to enable better search results[84]. Furthermore, all entities of the VbDMF model can be used as search criteria. Thus the query service is flexibly extensible to view model changes. In Section 7.4 we present our lightweight query language used by our prototype implementation in order to query view models and view model instances.

Registration service Via the registration service, developers can register new view models/view model instances and adapt existing view models/ view model instances. Still, we use view model instances and views as synonyms. As shown in Figure 7.3(a), the models and views are validated. In more detail, the views are validated against their view models and the view models are validated against their view meta-models. After successfully validating the views and models, they are stored in the DAS repository. In case of registering view model instances, the registration service can invoke the build/

deploy service in order to being able to deploy and test the DAS view model instances on a certain DAS provider.



(a) DAS Repository Registration Service (b) DAS Repository Publication Service

Figure 7.3: Registration Service and Publication Service

Build/ Deploy service By using the build/ deploy service, DAS can be deployed on a DAS provider. As shown in Figure 7.2, the passed view model instances can be transformed to source code by a view-to-code transformator. Afterwards, by a build process, the source code can be transformed to runnable code running on a DAS provider. When developing DAS, developers should firstly register and deploy the service via the registration service. After successfully testing the DAS, they should publish the DAS to other users and repositories via the following publication service.

Publication service The view models and view model instances can be published using the publishing service. During the publishing process, the view model instances can be registered to other repositories as well as to selected persons, teams, departments, or companies. The following Figure 7.3(b) depicts a basic activity workflow of our publication service implementation. Moreover, it illustrates the relationships between the DAS repository and the service repository: If the view is a service-related view that is part of the VbMF, the view will have to be registered to the service repository. In this case, the publication request is delegated to the synchronization service. Likewise, the data-related views of VbDMF can be synchronized to a schema repository [16]. Again, the service-related views of VbMF describe the DAS whereas the data-related views of VbDMF describe the underlying DAOs, ORMs, data objects, physical data, and database connection data. After this synchronization process, the DAS repository's internal view model instances are marked as synchronized and published.

Synchronization service The synchronization service is used to replicate view model instances between the DAS repository and other repositories. Likewise, related data from the other repositories need to be transferred to the DAS repository. This synchronization is only done for view model instances, but not for view models. Again, we do not need to store the view models redundantly, because they are only DAS repository internally used basically in order to specify our DAS repository view model instances.

During the synchronization process, a view model instance version is transferred to other repositories that are connected with the DAS repository. In case of the UDDI service repository, we can store the versioning information in the *instanceDetails* entity of the UDDI information model[25]. The *instanceDetails* entity is specified to store more detailed information of a service and is related to the *tModelInstanceInfo* entity of the basic *tModel* entity. Furthermore we support change log metadata about which user has inserted or updated a certain repository model and view alternatively.

Above we have described the services from an architecture point of view. For how to use the services from the user's point of view please refer to our tooling Section 7.4. Moreover, in our case study Section 7.5 we apply the DAS repository services using concrete use cases.

The DAS Repository View Model

In Chapter 4, we have already given a basic overview of VbDMF. In this section we present our DAS Repository View model with a decisive goal in the context of this thesis: by supporting introspection of DAS model instance data, underlying DAO model instance data, dependent ORM-specific model instance data and database configuration model instance data, it enables to bridge the gap between these data in order to improve documentation and maintainability.

In Figure 7.4 we display the relationships between the model entities and their related VbMF and VbDMF views in the DAS repository context. In the following we go deeper into describing the model entities of the DAS Repository View model. As already mentioned before, our VbMF/ VbDMF can be used to model arbitrary DAS implementations. However, in this thesis, we use DAOs as exemplary DAS implementation for object-oriented environments.

- The *Database Connection* class comprises a list of connection properties e.g. database url and name. So we can query all DAS from the DAS repository that belong to a database running on a certain location such as a host system. Furthermore, a *Database Connection* consists of zero or more *Tables* that in turn holds a list of *Columns*. These relationships allows us to query the DAS repository for all database operations that read or write certain tables or columns.

- In object-oriented programming languages information is stored in data object member variables. Our data model conforms to the object-oriented paradigm and contains a class *Data Object Type* which holds a list of *Member Variables*. As DAOs typically use ORM frameworks to map data object types to database tables, our

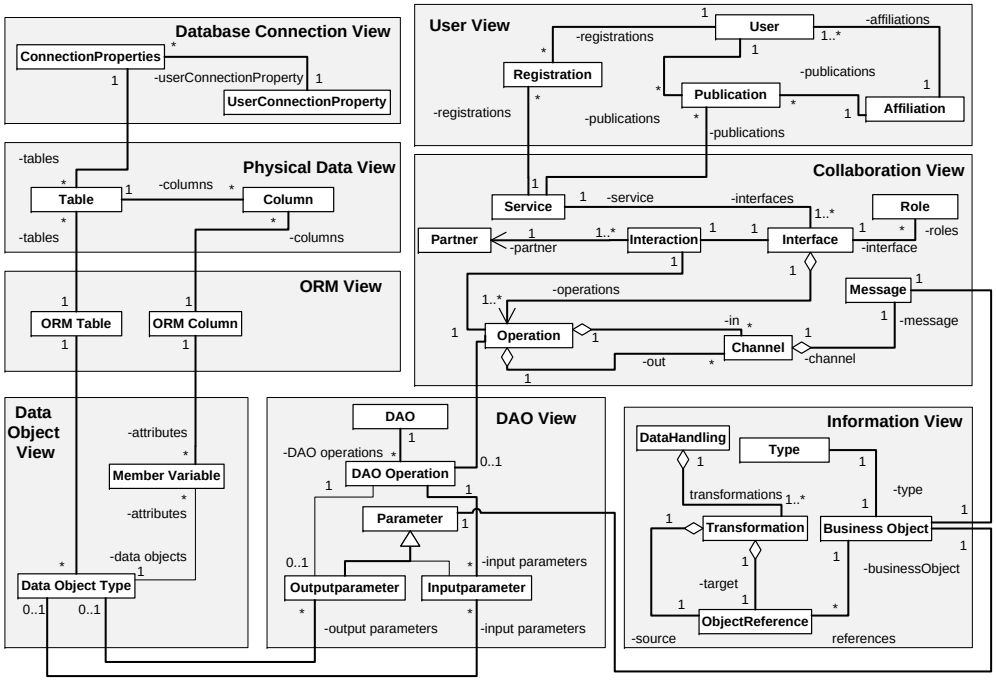


Figure 7.4: DAS Repository View Model

model provides an object-relational mapping of the *Data Object Type* to a (*Table*) using the mapping class *ORM Table*. The mapping class *ORM Column* allows for a more specific mapping between *Member Variable* and *Column* of a table. Using this additional ORM-specific information, we can generate the DAO source code. Furthermore we can retrieve all DAS/ DAOs that are based on a certain ORM framework.

- Each *DAO* consists of one or more *DAO Operations*. Each *DAO Operation* holds an attribute that stores the type of SQL statement (select, insert, update, delete). Furthermore it holds an *Output Parameter* and a list of *Input Parameters*. A parameter (*Input Parameter* or *Output Parameter*) can either be associated with a *Data Object Type* or with a simple type. A simple type is modeled as an attribute in the superclass *Parameter*. As a consequence of these relationships, we can query all DAO operations from the DAS repository that read or write certain data object types and member variables.

- The *Parameter* class of the DAO View is associated with a *Business Object*, of the Information View. Each *Business Object* has a *Type* and is an integration point, that can be used to combine a specified Collaboration View with an Information View and with a DAO View respectively. *Business Objects* of the Information View might go through some *Transformations* that convert or extract existing data to

form new pieces of data. These *Transformations* are performed inside a *DataHandling* object. The source or the target of a transformation is an *ObjectReference* class that holds a reference to a certain *BusinessObject*. The *Business Object* can be combined with the *Message* class of the Collaboration View model. The *Message* class basically specifies a message, described by a service description language e.g.[140]. The details of the *Message* class such as parameter types are not defined in the Collaboration View but by a *Business Object* of the Information View. Therefore the *Message* class becomes an integration point and can be combined with a *Business Object* of the Information View.

- In the Collaboration View model, the Service class exposes a number of *Interfaces*. Each Interface provides some *Operations*. An *Operation* represents an action that might need some inputs and produces some outputs via correspondent *Channels*. Each *Channel* holds a reference to a *Message* class. When an *Operation* of the Collaboration View is related to a *DAO Operation* of the DAO View, the *Operation* class acts as an integration point of the Collaboration View and the DAO View. The *Operation* class of the Collaboration View specifies the operation from the service point of view, whereas the DAO View specifies *DAO Operations* from the DAO point of view with its DAO input and output parameters.

- As shown in Figure 7.4, each *Service* holds a list of *Registrations* and *Publications*. The *Registration* class has a *n:1* relationship with the *User* class because a user typically registers more than one DAS at a time. After registering a DAS, the user can publish it. As shown in the data model, the class *Publication* has a n:1 relationship with the class *User* and with the class *Affiliation*, respectively. Thus, authorizations for a certain publication can be given to affiliations or/and users. The class *Affiliation* can consist of zero or more *User* classes.

7.4 Tooling: The View-Based Repository Client

Our view-based repository client prototype, shown in Figure 7.5, has been implemented as an Eclipse Plug-in to comfortably supporting developers in modeling DAS. In the following we describe our view-based repository client in more detail.

Using the View-Based Repository Client

The view-based repository client accesses the DAS repository by invoking its services. The UML activity diagram, displayed in Figure 7.6, illustrates the interaction of the view-based repository client and the DAS repository in more detail. We present a typical activity flow performed by stakeholders when modeling new or adapting existing models. Our view-based repository client, depicted in Figure 7.5, consists of several Eclipse views. In order to connect the Eclipse views in Figure 7.5 to the activities of Figure 7.6, the Eclipse views and the activities are labeled with corresponding numbers. In the following,

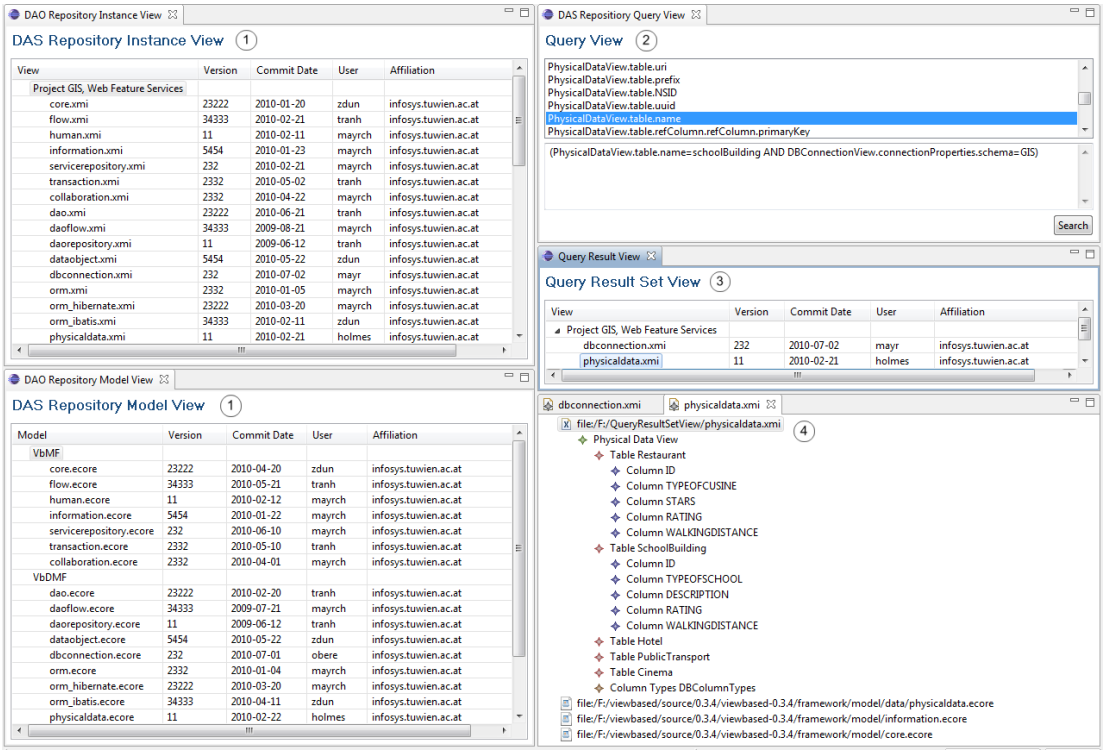


Figure 7.5: View-Based Repository Client GUI (Eclipse Plug-in)

we describe each of the depicted activities from the view-based repository client's point of view.

- *Retrieve views/ models manually:* Usually, if stakeholders are not firm with the DAS models and model instances, they first will have to acquaint themselves with the models of the *Eclipse DAS repository Model View (1)*, before they search for a specific view. The *Eclipse DAS Repository Model View (1)* lists all view models stored in the DAS repository. Stakeholders can find views and view models manually by traversing the *Eclipse DAS Repository Instance View (1)*, that lists all views stored in the DAS repository.

- *Retrieve views with search criteria:* Alternatively, in order to more comfortably query views for reuse, stakeholders can use the *Eclipse Query View (2)* that provides a query editor to express simple queries. On submit, a service operation request is sent to the DAS repository query service. As a result, a response with zero or more DAS repository views is returned. With our prototype implementation, the views are delivered as SOAP attachments from the DAS repository to the view-based repository client. The resulting view is an Ecore XMI model instance[128]. Up-to-now, our prototype view-based repository client only supports

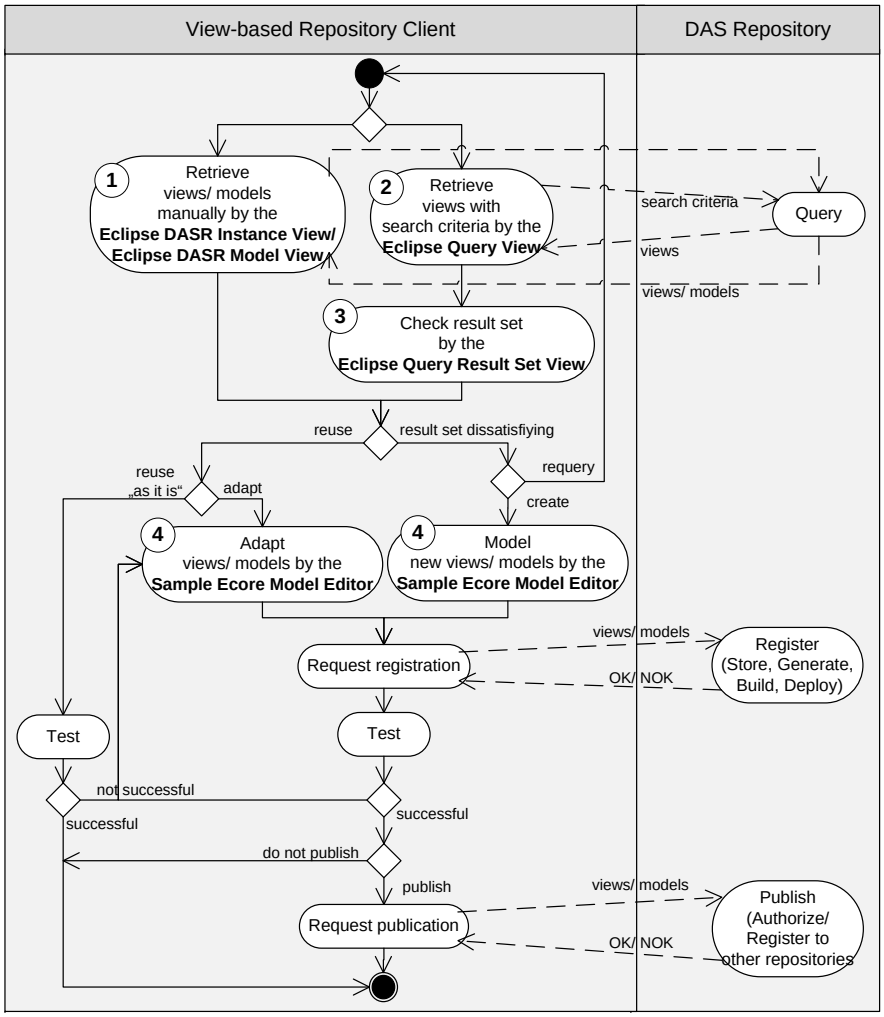


Figure 7.6: Activity Flow Diagram from the User's Point of View

structured querying for views, because, from our experience in larger enterprises, the number of view models is normally much lesser compared to the number of views ($< 0.5\%$). In order to quantify the number of view models and view model instances in the DAS repository, as shown in Figure 4.2, in the current version of VbMF/ VbDMF, a DAS can be described using 12 views, and each view is described by a view model. Thus, in order to describe 500 DAS, 6000 views and 12 view models have to be stored in the DAS repository. In this case, the number of view models stored in the DAS repository accounts for 0.5 % of the views.

- *Check result set:* After the views have been retrieved from the DAS repository, stakeholders such as developers can consider and check the result set in the *Eclipse*

Query Result Set View (3). The view-based repository client features the *Eclipse-embedded Sample Ecore Model Editor (4)* in order to view models and views. When a desired view that best possibly meets their requirements is displayed within the result set, developers can reuse it. Otherwise they either have to create a new view by using the *Sample Ecore Model Editor (4)* or to search again for suitable views using the *Eclipse Query View (2)* or the *Eclipse DAS Repository Instance View (1).*

- *Reuse:* In order to reuse a view, developers have two possibilities:
  - *Adapt views/ models:* In case of error corrections or to meet changing requirements, it may be necessary to adapt a view/ model. Editing existing views and models can be comfortably done by the *Eclipse-embedded Sample Ecore Model Editor (4).*
  - *Reuse 'as it is':* In case of developers find a desired view/ model, they can reuse it without adapting it.

- *Model new views/ models:* If developers do not find a suitable view/ model, they can model a new view/ model according to their requirements. An approach to develop new models from user requirements and to apply model transformations as a a base for the implementation can be found in [86]. The view-based repository client features the *Eclipse-embedded Sample Ecore Model Editor* that allows developers for comfortably specifying new view model instances. If developers do not intend to model their own models, they can re-query for a desired view by the *Eclipse Query View (2)*, the *Eclipse DAS Repository Instance View (1)*, and the *Eclipse DAS Repository Model View (1)*, respectively.

- *Request registration:* Developers can register new DAS/ DAO models and views by sending a service operation request with a model/ view as a SOAP attachment to the DAS repository (see Figure 7.2). Registering a model or view is possible by right-clicking on the context-menu within the *Eclipse-embedded Sample Ecore Model Editor (4).* Views neither adapted nor created are not subject for registration. After registering the DAS/DAO views/ models, they are persistently stored in the DAS repository. We use the Xpand language of the Eclipse M2T project [129] for source code generation from the defined model instances. The DAS themselves are generated from the Information View, the Collaboration View, and the Core View. The DAOs are generated from the various VbDMF views, the DAO View, the ORM View, and the Data Object Type View. As the DAO interfaces contain no ORM details, DAO interfaces are automatically generated simply from the DAO View and the Data Object Type View.

- *Test views/ models:* In order to test the views, they have to be be deployed on a test environment. After successfully testing the views, the deployed DAS can be invoked by a business process application. In order to test the view models, developers can specify test cases written in a conceptual schema testing language [131].

If the test fails, developers will have to re-adapt the views and models in order to fulfill their test requirements for the views/ models. Afterwards, they can be published to other users and repositories. If the test was not successful, the views/ models would have to be adapted to meet the test quality criteria. In this case, consequently, the views and models have to be both re-registered and re-tested.

- *Request publication:* After registering a DAS/ DAO, it can be published to selected persons, teams, departments, or companies so that they can query them. After successfully publishing the views and models, the authorized employees can view these DAS/ DAO views and models in the *Eclipse DAS Repository Instance View (1)* and in the *Eclipse DAS Repository Model View (1)*, respectively.

Using the Query Service

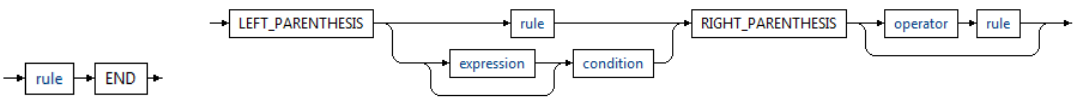
Finding models and model instances is a key functionality of the DAS repository. Hence, in this section we focus on how to use the query service from the view-based repository client's point of view. For this, we, in particular, define the query language used by our prototype implementation and secondly we analyze further support of a view-based repository client required when querying the DAS repository.

Query Language

Traditional database query languages, such as SQL and XQuery, are highly expressive but hard to learn. On the contrary, keyword queries are easy to use but lack the expressive power[24]. We chose to define our own language to query views and models by certain key word elements. Our lightweight technology-independent query language does not require stakeholders to be familiar with the specific characteristics of the underlying modeling language. Likewise, in order to search for views, stakeholders need not to be up-to-date with the Ecore meta-model elements. Though, they need an overview of the view model elements and the relationships between them. For this, in the following Section 7.4, we provide some GUI support.

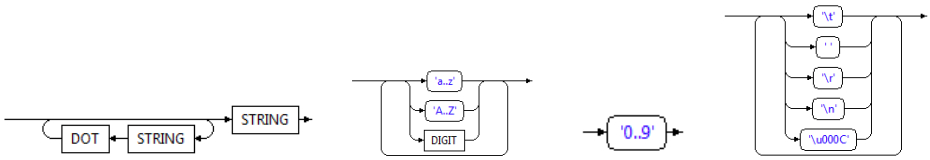
We developed our Query Language using the Eclipse ANTLR Plug-in[8]. Figure 7.7 contains a visual presentation of our Query Language in BNF notation. Rules displayed with a upper-case label are lexer rules, whereas the lower-case labeled rules are parser rules.[8]. As shown, our language consists of simple conditions (see Figure 7.7(d)) and boolean operators (see Figure 7.7(e)). As illustrated in Figure 7.7(c), these simple conditions and operators can be used in sequence within an expression. An expression in turn can be nested within a rule (Figure 7.7(b)). Finally, the rule rule_with_end of Figure 7.7(a) contains a rule and represents the root rule of our Query Language Definition to be verified.

When such a query is transmitted to the DAS repository by the query service, the service can return the relevant views and models matching the query. This lightweight query language is very simple, requires minimum of training effort and fulfills the requirement to find views and models by different search criteria.



(a) rule_with_end (b) rule

(c) expression (d) condition (e) operator



(f) MODELELEMENT_KEY (g) STRING (h) DIGIT (i) WHITESPACE

Figure 7.7: Query Language Definition in BNF Notation

Further support: Model Element Generator

In order to query views from the DAS repository by different search criteria, stakeholders need to know the view model elements. A common problem when querying reusable objects lies in the handling of misspelled words, synonyms and semantically equivalent words[110]. One way of addressing these issues is to limit the vocabulary, and to only allow queries drawn from this simple controlled vocabulary[81]. Hence, our DAS repository client proposes using view model elements for querying. The following Table 7.1 shows an extract of these view model elements. These view model elements are generated by Function KeywordGenerator. The function recursively steps through a view and outputs all elements from the view model itself, from its parent view models, and from its view model references. The output gives an overview about all possible key words within a view model.

Table 7.1: Model Element Generator: Extraction of Result Set

Key word
PhysicalDataView.table.prefix
PhysicalDataView.table.name
PhysicalDataView.table.column.primaryKey.name
PhysicalDataView.table.column.type.name
DaoView.dao.type.prefix
DaoView.dao.type.name
DaoView.dao.daoOperations.name
DaoOperations.daoOperation.inputParameter.name

```
   Input: Document model
   Input: Node currentNode
   Input: String output
 1 if (isEClassifier(currentNode)ORisRootNode(currentNode)) then
 2 |  HashMap < String, Node > hashMapAllNodes =
   |  getChildNodesAndInheritedChildNodes(model, currentNode);
 3 |  StringoutputPrefix = output;
 4 |  foreach String elementName ∈ hashMapAllNodes do
 5 |  |  NodechildNode = hashMapAllNodes.get(modelName);
 6 |  |  if (NOTchildNode.getNodeName().equals(""#text"")) then
 7 |  |  |  if (NOTisReference(childNode)) then
 8 |  |  |  |  if (NOTisEClassifier(childNode)) then
 9 |  |  |  |  |  if (isRootNode(currentNode)) then
10 |  |  |  |  |  |  output = getNodeOutputString(outputPrefix, childNode);
11 |  |  |  |  |  end
12 |  |  |  |  |  KeywordGenerator(model, childNode, output);
13 |  |  |  |  end
14 |  |  |  |  else
15 |  |  |  |  |  output = getNodeOutputString(outputPrefix, childNode);
16 |  |  |  |  |  System.out.println(output);
17 |  |  |  |  end
18 |  |  |  end
19 |  |  |  else
20 |  |  |  |  output = getNodeOutputString(outputPrefix, childNode);
21 |  |  |  |  NoderefNode = getReferenceNode(modelName, childNode);
22 |  |  |  |  if (NOTrefNode == null) then
23 |  |  |  |  |  KeywordGenerator(getModelByName(modelName), refNode, nodeOutputName);
24 |  |  |  |  end
25 |  |  |  end
26 |  |  end
27 |  end
28 end
```

Function KeywordGenerator

In contrast to extracting elements from view models, extracting model elements from the underlying Ecore meta-model of the Eclipse Modeling Framework (EMF) [128] is much simpler. The reason for this is, that the elements have to be only extracted from one Ecore meta-model instead of from several related view models.

7.5 Case Study

In this case study we show how our approach can be applied to geographic information systems (GIS) [79]. GIS are large-scale information systems making huge amount of spatial as well as non-spatial data available over the internet [135]. A GIS data model usually consists of a large number of entities [79]. There are several international standards produced by the ISO/TC 211 group that describe data models for geographic information, information management and information services. In particular, the ISO 19119 specification [102] provides a framework for specifying individual geographic information services. On top of this specification, the open geospatial consortium (OGC) establishes several OGC web service (OWS) standards for spatial data. One example is the OGC web feature service (WFS) specification [103]. WFS allow a client to retrieve and update spatial and non-spatial geographic data, encoded in geography markup language (GML) [105], an XML grammar for expressing geographical features. Such geographical features are e.g. restaurants, hotels, sights, indoor swimming pools, cinema, schools, gas stations, shops etc. Examples of spatial data include coordinates, height and width. Examples

of non-spatial data are the school building type or the average water temperature of indoor swimming pools. According to the OpenGIS WFS implementation specification [103], each WFS basically provides three operations: The operation *GetCapabilities* indicates serviceable feature types, the operation *DescribeFeatureService* provides the structure of serviceable feature types, and the *GetFeature* operation enables querying of certain feature instances by spatial and non-spatial search criteria. Optionally a WFS can provide a *Transaction* operation in order to service feature modifications.

Applying our Approach to Web Feature Services

In the following we apply our architecture approach to WFS in order to enhance documentation, traceability and maintainability of data access of WFS.

At first, there is a need to relate WFS terminology to the DAS terminology of this thesis:

1. WFS vs. DAS: A WFS can read and write spatial and non-spatial data from an RDBMS. Consequently, we define WFS as specialization of DAS.

2. Web Catalogue Service vs. Service Repository: The service repository, defined in this thesis, manages DAS metadata and provides a query service enabling business process applications to find suitable DAS by different search criteria. Accordingly, a web catalogue service is a service repository managing spatial and non-spatial WFS metadata.

3. WFS Provider vs. DAS Provider: Whereas DAS are available on a DAS provider, the more particular WFS are deployed on a WFS provider.

As shown in Figure 7.8, business process applications can find suitable WFS by querying a web catalogue service [104]. The web catalogue service manages WFS metadata enabling business process applications to retrieve WFS by diverse search criteria. After having found a suitable WFS from the web catalogue service, the business process application can invoke this WFS. Each WFS can either process a request by its own or it delegates the request to another WFS. In order to process a request by its own, the WFS usually reads or writes spatial and/or non-spatial data from a geographic database.

WFS operations such as the *GetFeature* operation, can handle diverse feature requests. In contrast, a DAS operation is more proprietary by processing one single data access request. Moreover, a WFS will be able to delegate a request to another WFS, if it cannot fulfill the request itself. As a result, we have to extend our VbDMF model by defining new view model elements for WFS. In the following we illustrate the necessary steps to extend VbDMF with new WFS view models and to create view model instances from these new views:

- *Model new views/ models:* By using the *Eclipse-embedded Sample Ecore Model Editor*, developers can comfortably specify new WFS view models and views. In the following, we extend VBDMF by creating a new view model, the WFS Information

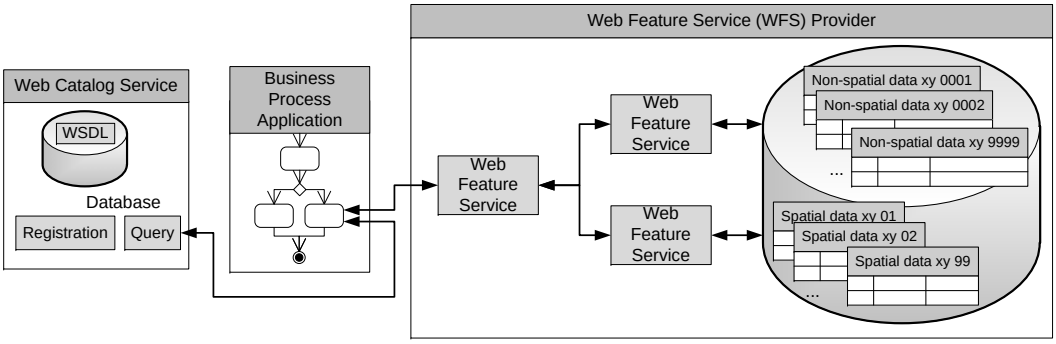


Figure 7.8: Case Study: Applying WFS to the VMDA

View, shown in Figure 7.9, describing specific WFS features. This WFS Information View consists of a WFS *Feature Type List* that is derived from the *Business Object* entity of the Information View. Hence, like a *Business Object* entity, a *Feature Type List* entity corresponds to a *Parameter* of the DAO View. Each *Feature Type List* consists of a list of *Feature Type* entities. Each *Feature Type* in turn comprises zero or more *Property Type* entities and is related to a *Parameter* entity of the DAO View. As WFS are able to delegate requests to another WFS, each *Feature Type* entity is related to zero or one *Service Operation* entities. Based on the VbMF and VbDMF view models, stakeholders can model new views in order to describe specific WFS. During the modeling process the DAS repository's view-to-code transformator has to be extended in order to generate source code from the specified WFS views. Moreover, the queries of a WFS feature request need to be mapped to the SQL-based DAO operations[138]. This WFS Query-to-DAO translation needs to be part of the resulting source code.

- *Request registration:* Developers can register the newly created WFS views/ models by sending a service operation request to the DAS repository. After registering the WFS/DAO views/ models, they are persistently stored in the DAS repository. As a result, the view-to-code transformator generates WFS source code from the views. Furthermore, the DAS repository's build/ deploy service builds the WFS source code and deploys the resulting WFS on a certain WFS provider.

- *Test views/ models:* Once the newly registered views and models have been successfully tested on the WFS provider, they can be published.

- *Request publication:* After successfully registering new views/models, they can be published to selected persons, teams, departments, or companies (see Figure 7.2) so that they can query them.

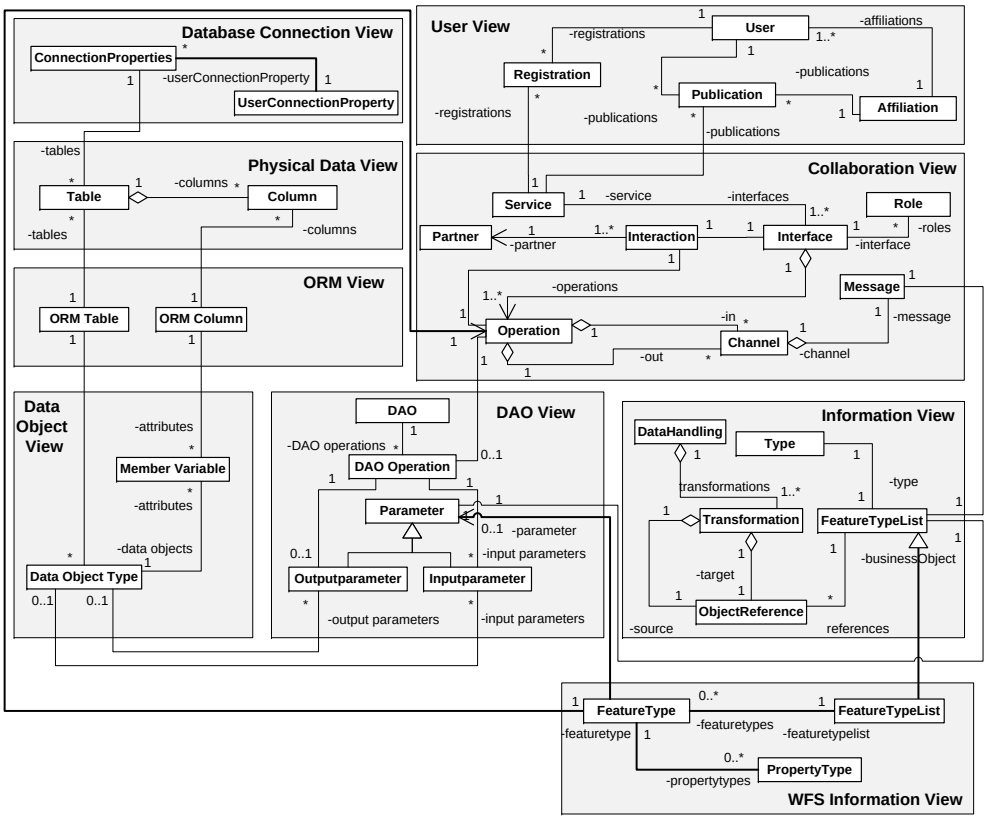


Figure 7.9: Case Study: Extending VbDMF by a New WFS Information View

Making Use of the DAS Repository

GIS usually have to manage a very large number of WFS. Moreover, each WFS consists of different features with specific spatial and non-spatial feature attributes. These feature attributes are stored in a geographic database. During our studies, we detected a documentation gap between the DAOs, the underlying object-relational mappings and the underlying data storage schemes. This makes it difficult for stakeholders such as system architects and database administrators to inspect the relationships between these different layers. As we provide a full-blown model of the WFS, we can use this information to document the relationships between the data storage schemes, the DAOs, and the WFS. Moreover, our approach provides the basis to view relevant parts of a WFS tailored to the requirements of the stakeholders at first sight. By using our view-based repository client, stakeholders can search for particular WFS by different search criteria such as tables, columns, data objects etc.:

1. *Retrieve views with search criteria:* Developers can look for WFS by diverse search criteria, e.g. they can query WFS that read or write a certain database table of

certain database schema. For this purpose, stakeholders can write a query within the view-based repository client's Eclipse Query View, in example: *(PhysicalDataView.table.name=schoolBuilding AND DBConnectionView.connectionProperties.schema=GIS)*. After entering the submit button, the view-based repository client invokes the DAS repository query service.

2. *Check result set:* As a result, as shown in Figure 7.5, a DBConnectionView and a PhysicalDataView matching the search criteria are returned and can be viewed in the Eclipse Query Result View. The developer can acquaint himself with the WFS and, for example, contact the WFS owner. If the desired PhysicalDataView or DBConnectionView is in the result set, the WFS can be reused. Otherwise, a new search can be started.

Next, we show, how the DAS repository's query interface and the view-based repository client can be exploited in order to selectively adapt existing views. Let us assume that the database connection settings of several WFS change, e.g. because the underlying geographic database is transferred from one server to another server. Usually, these database connection settings are part of the WFS/ DAO source code. With our approach, database administrators can adapt the settings without help of the WFS/ DAO developers by performing the following steps:

1. *Retrieve views with search criteria:* Database administrators can ask the DAS repository's query service via the view-based repository client, if there exist any WFS accessing a database running on a certain server. Hereto, they type the following query in the Eclipse Query View:
(DBConnectionView.connectionProperties.server=http://192.168.1.11:8080)

2. *Check result set* Afterwards, the query service returns a list of DBConnectionView views matching the query.

3. *Adapt views:* The database administrators use the view-based repository client to adapt the existing view model instances. By using the Eclipse-embedded Sample Ecore Model Editor, they can change the value of the *DBConnectionView.connectionProperties.server* model element from *http://192.168.1.11:8080* to *http://192.168.1.12:8080*.

4. *Request Registration:* Afterwards, database administrators have to register, build and deploy the adapted WFS by invoking the register service with the new DBConnectionView as SOAP attachment.

5. *Test:* Then, the WFS with the newly configured database connection need to be tested.

6. *Publish new DAOs:* After registering the successfully tested WFS, database administrators publish the WFS views to the same group as before.

Hence, as by using DAS, by using WFS, applications can read and write data from a higher level than the DAO layer. When an underlying technology changes, concerned stakeholders can focus on the specific view and perform the adaption without the need to involve other stakeholders. In the following Section 7.6 we illustrate further use cases in order to quantitatively evaluate our approach.

7.6 Evaluation

In this section we describe illustrative use cases in order to quantitatively evaluate our approach. In particular, each of these use cases looks for geographic web feature services (WFS) by certain search criteria. We have already introduced WFS of large-scale GIS applications in Section 7.5.

According to the analysis of Banker et al.[12] who analyzed the evolving repositories of two large firms, "programmers are willing to bear extremely low search costs before choosing to just write their own objects". Moreover, "the subject of quality engineering and management is about reducing the variability in products and processes, quality costs, and to provide maximum satisfaction to the customers through improved product performance"[91]. Accordingly, developers that are not fully positive about the performance of the DAS repository would not use it. Thus the response time of the DAS repository is one of the key non-functional requirements that need to be fulfilled, so that developers can gain the best-possible benefit from the DAS repository. As querying the DAS repository is the key functionality in our VMDA, we quantify the response time and the scalability of exemplary use cases invoking the DAS repository's query service.

There are two main approaches in order to map models to an RDBMS[84]: an XSD model mapping approach and a domain model mapping approach. According to the architectural decision in Section 6.3, for performance reasons, it is the best to use an XSD model mapping approach instead of a domain-based mapping approach. In our example, this means, we should use an XMI-based mapping approach to map Ecore elements to database tables. However, we will show that even a common domain-model-based approach shows acceptable performance results. Thus, each entity of the VbDMF models, displayed in Figure 7.4, is physically mapped to a database table. This basic mapping from entities to tables is done both for the view model entities and for the Ecore meta-model entities, so that we can both support structured querying for views and view models.

In the following, we solely describe the cases frequently fulfilled by stakeholders developing and maintaining geographic WFS when using our DAS repository. These use cases firstly result from our study of analyzing persistent data access in service-oriented environments in a large enterprise and secondly from analyzing WFS of commercial[35] and open-source[82] GIS. The use cases demonstrate how the relationships between the WFS, the DAOs, and the data storage schemes can be exploited to query persistent data access elements from the DAS repository.

A Java-based test client application performs each use case by invoking the DAS repository's query service by given search criteria. After invoking the query service,

our test client application receives the according result set. As we want to test the scalability of our repository, for each exemplary use case, we invoke different query service implementations, each accessing a certain repository database of 10, 100, 1000, 10000, and 100000 WFS view model instances respectively. We assume that each WFS view consists of 10 WFS features. After each service invocation we restart the MySQL Server service to avoid caching effects during our measurements.

Use cases For each use case, we describe the table joins performed by the query service. We choose one representative use case query for each number of table joins. Surely, there are other queries that might be useful to find appropriate WFS. However, we chose those use cases that, according to our experiences and studies, are most likely to be used by developers.

- Use Case Query 1 (Query by Database): If database administrators intend to migrate a database from one database server to another server, they can use a query by database connection to find all WFS modeling a relation to this database connection. This type of query does not require a table join, because only the *Database* table is selected.

- Use Case Query 2 (Query by Table, Column): If DAS developers need to know which WFS access a certain column of a table, DAS developers can ask the DAS repository. In order to perform this query, joining the *Table* and the *Column* table is necessary. The two conditions should be concatenated with the boolean AND operator.

- Use Case Query 3 (Query by Database, Table, Registration): In case a specific database table fails, stakeholders such as system architects or database administrators might want to inform the relevant WFS providers about this failure. However, only some WFS providers have registered to be kept informed of system failures. In order to find the resulting WFS, the query service joins the tables *Database*, *Table* and *Registration*. Again, the conditions could be concatenated with the boolean AND operator.

- Use Case Query 4 (Query by FeatureType, PropertyType, ORM Table, ORM Column): DAS developers use this query to understand which WFS features access which database tables. In order to process this query, DAS developers insert both the name of a certain *FeatureType* and of a certain *PropertyType* into the search condition. In example, the feature type could be a bus station and the property type could be a spatial data type that describes the distance from the current position. The tables *ORM Table* and *ORM Column* define the mapping between a *FeatureType* and a *Table*, and the mapping between a *FeatureProperty* and a table *Column*, respectively. Thus, these mapping tables should also be included into the join. The specific search conditions should be linked with the boolean AND operator.

- Use Case Query 5 (Query by Registration, Publication, User, Affiliation, Feature Type): By using this query, developers can find WFS, they have registered or published within a certain time period. In particular, they use this query, when they only know the name of the registered and published feature and the month of registration/ publication date. Thus, developers have to find all published or registered WFS at a certain date by a certain user of a certain affiliation with a specific feature. Hereto, both the boolean OR and AND operator can be used to join the tables involved, namely the *Registration* table, the *Publication* table, the *User* table, the *Affiliation* table and the *FeatureType* table.

- Use Case Query 6 (Query by DAO, DAO Operation, Output Parameter, Input Parameter, Data Object Type, ORM Table): Database developers and database administrators can use this query in order to document which DAO operations access which database tables. For this purpose, stakeholders specify the name of the *DAO*, the *DAO Operation*, the *Input Parameter*, the *Output Parameter*, and the *Data Object Type*. In order to map a data object type to a specific database table, the *ORM Table* table has also be included into the join. As a result, they get the database tables accessed by this DAO operation.

- Use Case Query 7 (Query by User, Registration, Publication, Affiliation, FeatureType, Operation, PropertyType): In addition to Query 5, stakeholders can add the *Operation* table as additional search criteria in order to search for WFS with specific operations e.g. for transaction WFS, that support the transaction operation. Moreover, if stakeholders know the specific *PropertyType* of a *FeatureType* of features they have registered and published, they can add the *PropertyType* table to the search condition by using the boolean operator AND or OR.

- Use Case Query 8 (Query by DAO, DAO Operation, Output Parameter, Input Parameter, Data Object Type, ORM Table, Member Variable, ORM Column): Developers use this query in order to find the database columns mapped to a certain member variable as part of a certain DAO operation. In addition to Use Case Query 6, stakeholders can add the *Member Variable* table as additional search criteria. In order to map member variables to table columns, the query service has to include the *ORM Column* table in the join.

- Use Case Query 9 (Query by User, Registration, Publication, Affiliation, FeatureType, Operation, PropertyType, Database, Table): If, in addition to the search criteria in Query 7, the name of the database and table is known, developers can put both the *Database* table and *Table* table into the table join.

Test requirements Table 7.2 shows the numbers of data rows imported into each of the tables used for the measurement. According to Table 7.2, we define that a WFS consists of 5 operations and of 10 *Features*. Each *Feature* comprises 10 *Feature Properties* and corresponds to one *DAO*. We further assume the simple case that each *DAO* accesses one *Table*. An *ORM Table* maps a *Table* to exactly one *Data Object Type*. A *Table*

Table 7.2: Experiment: Number of Table Rows Related to Number of WFS

Table	10 WFS	100 WFS	1000 WFS	10000 WFS	100000 WFS
Affiliation	20	200	2000	20000	200000
Column	1000	10000	100000	1000000	10000000
DAO	100	1000	10000	100000	1000000
DAO Operation	500	5000	50000	500000	5000000
Data Object Type	100	1000	10000	100000	1000000
Database	2	20	200	2000	20000
Feature Type	100	1000	10000	100000	1000000
Input Parameter	500	5000	50000	500000	5000000
Output Parameter	500	5000	50000	500000	5000000
Member Variable	1000	10000	100000	1000000	10000000
Operation	50	500	5000	50000	500000
ORM Column	1000	10000	100000	1000000	10000000
ORM Table	100	1000	10000	100000	1000000
Property Type	1000	10000	100000	1000000	10000000
Publication	1000	10000	100000	1000000	10000000
Registration	2000	20000	200000	2000000	20000000
Table	100	1000	10000	100000	1000000
User	10	100	1000	10000	100000

consists of 10 *Columns*. And *ORM column* maps each *Column* to exactly one *member variable*. Each *member Variable* corresponds to one simple *Data Object Type*. Each *DAO* contains 5 *DAO Operations*. Each *DAO Operation* in turn contains one *Input Parameter* and one *Output Parameter*. Each *Output Parameter* and each *Input Parameter* are always mapped to one possible complex *data object type* or a simple *Data Object Type*. Simple *Data Object Types* are disregarded in Table 7.2 because for our measurements the number of simple types is rather unimportant. We estimated that each *user* can belong to two different *affiliations* during their employee membership. For this test, we estimated further that a WFS is registered a 2000 times and published a 1000 times in average, and 10 *users* publish and register one specific WFS.

Table 7.3: Experiment Settings

Processor:	Intel(R) Core(TM)2 Quad CPU 2.4 GHz
RAM:	4 GB
Operating System:	Windows 7 (64 bit)
Database:	MySQL Server 5.1
Java Version:	1.6.0_10
MySQL Server Type:	Developer Machine

In Table 7.3 we summarize the settings of our test machine. Please note that our performance measurements are based on a fully normalized data model that does not contain any redundant column data.

Results We measured the response times for queries related to the number of WFS and the number of table joins necessary to perform the query. The following Figure 7.10 displays the use case queries on the x-axis and the response time in milliseconds on the y-axis. The resulting curve is approximately proportional with the number of table joins. However, the curve is not exactly linear. We think this non-linear functional behavior results (among others) from the following reasons:

- Different queries are optimized to a different degree

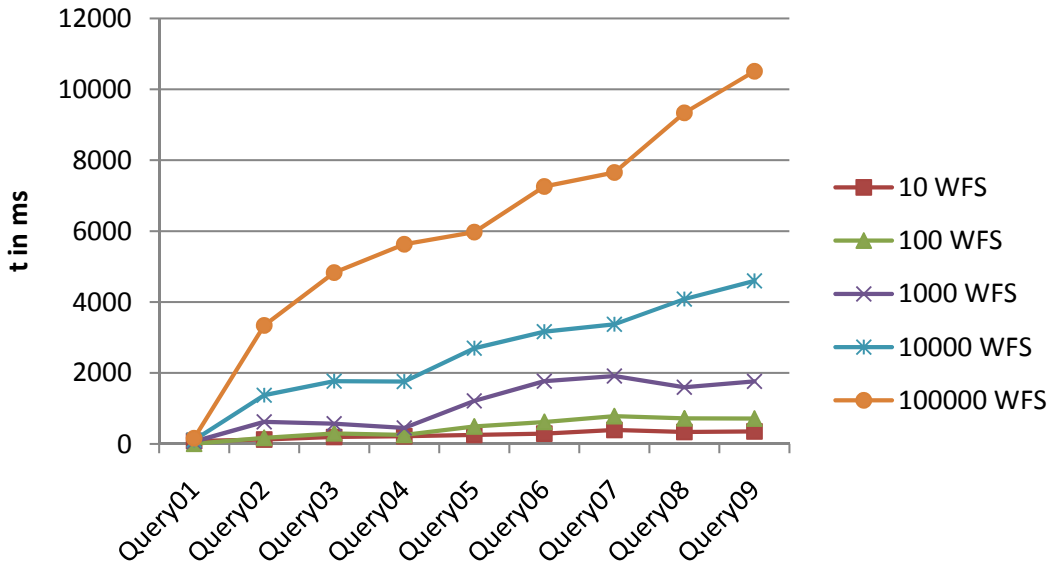


Figure 7.10: Experiment Result: Response Time Against Number of Table Joins

- Randomized search criteria can lead to faster or slower results
- Different queries with different tables joining various numbers of rows as described in Table 7.2

As shown in Figure 7.10, our repository offers acceptable response times even for a large number of WFS. To illustrate this, let us come back to our case study in Section 7.5. GIS usually manage several thousand features provided by several hundred WFS. As shown in Figure 7.10, the performance for querying views from 100000 WFS and 1000000 available WFS features in the repository is acceptable. Fortunately, in practice, the number of search criteria, and thus the number of table joins, is usually low (approximately 1-4), therefore, resulting in a very good overall performance.

Our approach scales well with increasing numbers of WFS. Figure 7.11 and Figure 7.12 show the query response times and the logarithmic query response times respectively with an increasing number of WFS. Again, the y-axis displays the response time in milliseconds. In contrast to Figure 7.10, the x-axis displays the number of WFS. As shown in Figure 7.12, from up to 1000 WFS, for all queries, the response time values increase virtually linearly with increasing number of WFS. The reason why querying 10 WFS results in quicker response times than querying 100 WFS is that indices are ignored by the database query optimizer when there are only very few rows in a table. For numbers of at least 1000 WFS the optimizer uses the indices to perform the queries.

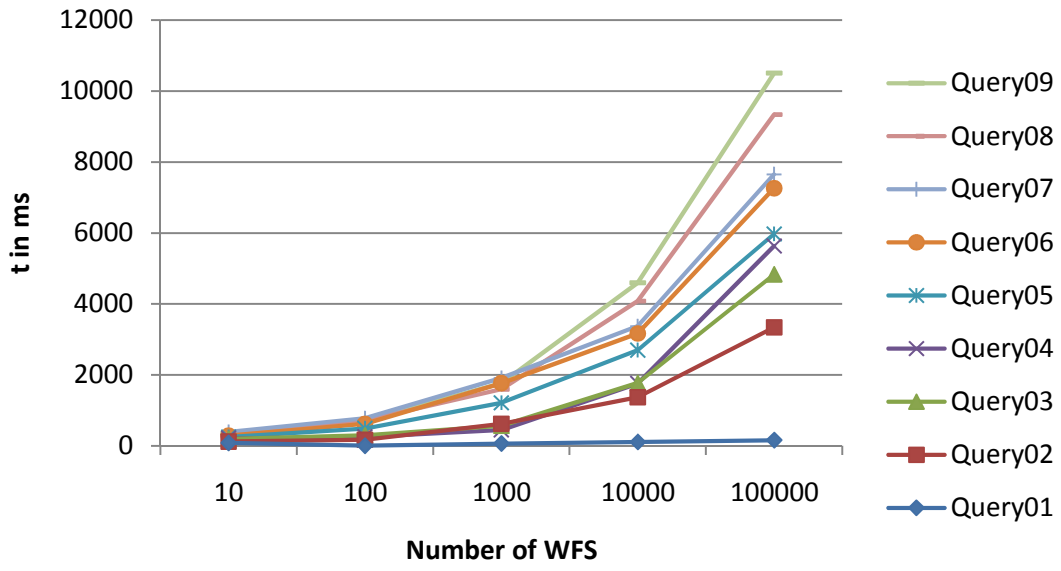


Figure 7.11: Experiment Result: Response time Against Number of WFS

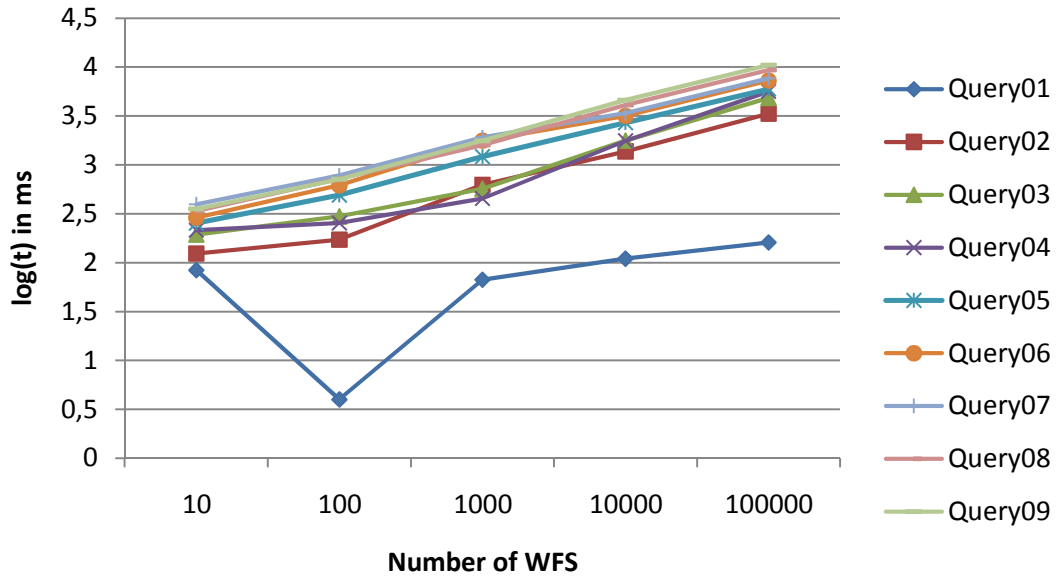


Figure 7.12: Experiment Result: Logarithmic Response Time Against Number of WFS

7.7 Summary

In this chapter, we introduced our view-based model-driven data access architecture (VMDA) for managing DAS in a process-driven SOA. Our approach improves documentation of the relationships between DAS, DAOs and data storage schemes and thus contributes to a higher software development productivity and maintainability. For example, our VMDA documents which user provides which DAS, which DAOs are related to the DAS and which database tables are read and written by the DAS and DAOs.

Moreover, as the number of software components grows, data development complexity increases with the number of DAS. So retrieving a particular DAS can be complex and time-consuming. In order to tackle these issues and to enable better maintainability, we specified a view-based model-driven data-access architecture (VMDA) managing the views, models and relationships between them. This service-oriented architecture is composed of well-known concepts such as a model-driven repository and a view-based repository client. Though, we combine these concepts in order to efficiently develop, maintain, trace and deploy DAS in a process-driven SOA. In order to reduce the complexity of managing DAS, we separate the DAS into different views, namely the Collaboration View, the Information View, the ORM View, the DAO View, the Physical Data View, the Database Connection View, and the User View. The DAS repository stores DAS models and views, and consists of several services providing basic functionalities to query, register, publish DAS. Due to the structured nature of the DAS views and models, we can query DAS by implementation, data storage schema, and metadata artifacts.

CHAPTER 8

Conclusion

8.1 Summary of the Research Problems

In this thesis we identified current problems arised when developing and maintaining persistent data access in process-driven SOA.

- Documentation lacks quality: One major problem, when developing and maintaining DAS, is the documentation gap between the data access services (DAS), the underlying DAS implementations such as data access objects (DAO), and the data storage schemes.

- Insufficient stakeholders support: Stakeholders need to focus on different concerns of persistent data access in a process-driven SOA. However, in hetereogenous development environments, it is not easy to concentrate on specific interests of persistent data access.

- No adequate tooling and language support for solving structural problems in business processes: Current BPMS and business process modeling languages do not provide proper support for solving structural business problems concerning persistent data access e.g. deadlocks.

- Unsatisfied reuse and maintainability of DAS: Locating a specific DAS within hundreds or thousands of DAS is a time-consuming task. In order to being able to efficiently reuse and maintain DAS, they need to be effectively managed. However, up-to-now, there exists no suitable DAS managing architecture to model, query, retrieve, and publish DAS.

8.2 Summary of the Contributions

Our contributions focus on better modeling and managing persistent data access. We provided a novel approach to improve maintaining, reusing, and tracing persistent data access in process-driven SOAs from the service provider's view. Our novel contributions are based on the following basic concepts:

- *Model-driven development* to improve documentation of persistent data access in process-driven SOAs
- *Views* to support different stakeholders to view their relevant piece of data access
- A *repository architecture* for better managing and maintaining persistent data access
- A *reusable architectural decision model* to document basic architectural design decisions
- *Persistent data access flows* to solve structural problems concerning persistent data access in business process

In the following we summarize our novel contributions:

View-based data modeling framework In this thesis, we introduced VbDMF, a view-based model-driven framework for modeling persistent data access in process-driven SOAs. With VbDMF we can document the relationships between data access services (DAS), the underlying data access objects (DAOs), the object-relational mappings (ORM), and the physical data storage schemes. Due to the model-driven approach, stakeholders can model persistent data access from a higher level than the source code level. In addition, due to the views concept, they can individually focus on their particular interests of persistent data access within the business process. Moreover, we can reuse the high-structured DAS for model-to-code and model-to-documentation transformations.

Persistent data access flows We applied our view-based modeling concepts to solve structural problems concerning persistent data access in business processes. In particular, in a number of use case scenarios, we illustrated, how persistent data access flows can be applied to solve structural problems in business processes concerning persistent data access. We present a novel view-based modeling solution to extract these persistent data access flows from whole business processes. The feasibility and applicability are shown in a case study. Finally, we evaluated the correctness and performance of the used algorithms.

Reusable architectural decision model (RADM) for model and metadata repositories In order to design and set-up model and metadata repositories, a series of architectural decisions have to be made. In this thesis, we presented a reusable architectural decision model (RADM) for model and metadata repositories. This RADM incorporates best practices in setting-up and developing model and metadata repositories. In a case study, we apply our architectural decisions to build a DAS repository managing VbDMF models and views.

View-based model-driven data access architecture (VMDA) We introduced a view-based model-driven architecture used to better manage, maintain, and reuse data access services (DAS). Our VMDA uses VbMF/ VbDMF to specify the models and views. In a large-scale case study, we show how our architecture approach can be applied to web feature services (WFS) in geographic information systems (GIS). With our performance evaluation, we proved the acceptable response times of the repository.

8.3 Future work

In this thesis we have presented a series of encouraging concepts to improve managing and modeling persistent data in process-driven SOAs. During our research we have touched many interesting research topics being worth further research investment, however being out of scope of this thesis.

Repository requirements In our approach we have provided a lightweight technology-independent query language to search for VbDMF views. Furthermore, advanced searching capabilities, such as those that can be provided on top of our approach, are desirable. The selective use of ontologies could improve the quality of the retrieved result set.

In addition, further work is necessary to coping with other important repository's requirements such as event notification, configuration control, and security.

Moreover, in order to synchronize data from other repositories to the DAS repository, a sophisticated data re-engineering is necessary, that is also part of our future work.

Runtime aspects In this thesis we focused on improving managing and modeling persistent data access at modeling time. Thus, runtime aspects such as dynamic invocation of DAS need to be discussed and defined. In particular, further work can include runtime statistics for measuring how often a DAS operation has been invoked, etc.

Reverse code engineering Furthermore, we will focus on source code re-engineering in order to being able to exploit our approach when no view model instances are available.

Tool support Working with models in tools still lacks refinement [70], we continue focusing on developing suitable tool chains for modeling persistent data access in process-driven SOAs. In particular, We increasingly concentrate on server-client interactions and repository client tools, that give value to the repositories.

Bibliography

[1] Xiaoming Liu 0005, Kurt Maly, Mohammad Zubair, and Michael L. Nelson. Repository synchronization in the OAI framework. In *JCDL*, pages 191–198, 2003.

[2] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Balancing push and pull for data broadcast. In *SIGMOD Conference*, pages 183–194, 1997.

[3] Asif Akram, David Meredith, and Rob Allan. Evaluation of bpel to scientific workflows. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 269–274, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In Rakesh Agrawal and Klaus R. Dittrich, editors, *ICDE*, pages 141–152. IEEE Computer Society, 2002.

[5] Eyhab Al-Masri and Qusay H. Mahmoud. Discovering the best web service. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1257–1258, New York, NY, USA, 2007. ACM.

[6] AndroMDA. EMF UML2 Repository. http://galaxy.andromda.org/docs-3.2/andromda-repository-emf-uml2/index.html, Nov 2006.

[7] AndroMDA.org. EJB3 Cartridge. http://www.andromda.org/docs/andromda-cartridges/andromda-ejb3-cartridge/, Copyright 2006-2011.

[8] ANTLR v3. ANTLR IDE. an eclipse plugin for ANTLRv3 grammars. http://antlrv3ide.sourceforge.net/, Retrieved June, 2010.

[9] Apache Software Foundation. Axis2/Java. http://ws.apache.org/axis2/index.html, 2004-2008.

[10] Mustafa Atay, Yezhou Sun, Dapeng Liu, Shiyong Lu, and Farshad Fotouhi. Mapping xml data to relational data: A dom-based approach. In *Eighth IASTED International Conference on Internet and Multimedia Systems and Applications, Kauai*, pages 59–64, 2004.

[11] Ahmed Awad and Frank Puhlmann. Structural detection of deadlocks in business process models. In *BIS*, pages 239–250, 2008.

[12] Rajiv D. Banker, Robert J. Kauffman, and Dani Zweig. Repository evaluation of software reuse. *IEEE Trans. Software Eng.*, 19(4):379–389, 1993.

[13] Philip A. Bernstein. Repositories and object oriented databases. In *BTW*, pages 34–46, 1997.

[14] Philip A. Bernstein and Umeshwar Dayal. An overview of repository technology. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 705–713, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[15] Shahid Nazir Bhatti and Asif Muhammad Malik. An XML-based framework for bidirectional transformation in model-driven architecture (MDA). *SIGSOFT Softw. Eng. Notes*, 34:1–5, May 2009.

[16] Hassina Bounif and Rachel Pottinger. Schema Repository for Database Schema Evolution. In *DEXA Workshops*, pages 647–651. IEEE Computer Society, 2006.

[17] BrainML. Neurodatabase Construction Kit, Repository Server. http://brainml.org, Retrieved April, 2011.

[18] Peter Brittenham. Web Services Inspection Language (WS-Inspection) 1.0. http://www-106.ibm.com/developerworks/webservices/library/ws-wsilover/#resources, June 2002.

[19] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[20] Mario Cannataro, Domenico Talia, and Paolo Trunfio. Distributed data mining on the grid. *Future Generation Computer Systems (FGCS)*, 18(8):1101–1112, 2002.

[21] J. Cardoso. Control-flow Complexity Measurement of Processes and Weyuker's Properties. In *6th International Enformatika Conference*, Transactions on Enformatika, Systems Sciences and Engineering, Vol. 8, pages 213–218, 2005.

[22] Michael J. Carey, Panagiotis Reveliotis, Sachin Thatte, and Till Westmann. Data service modeling in the aqualogic data services platform. In *SERVICES I*, pages 78–80, 2008.

[23] Silvana Castano, Alfio Ferrara, G. S. Kuruvilla Ottathycal, and Valeria De Antonellis. A disciplined approach for the integration of heterogeneous xml datasources. In *DEXA '02: Proceedings of the 13th International Workshop on Database and Expert Systems Applications*, pages 103–110, Washington, DC, USA, 2002. IEEE Computer Society.

[24] Yi Chen, Wei Wang 0011, Ziyang Liu, and Xuemin Lin. Keyword search on structured and semi-structured data. In *SIGMOD Conference*, pages 1005–1010, 2009.

[25] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. UDDI Version 3.0.2, UDDI Spec Technical Committee Draft. http://www.uddi.org/pubs/uddi_v3.htm, Oct 2004.

[26] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, pages 86–93, 2002.

[27] Java SE Technologies Database. The java database connectivity (jdbc). http://java.sun.com/javase/technologies/database/, 2001.

[28] G. Dedene and M. Snoeck. Formal deadlock elimination in an object oriented conceptual schema. *Data Knowl. Eng.*, 15(1):1–30, 1995.

[29] James Donahue. Integrating programming languages with database systems. In *Persistence and Data Types: Papers for the Appin Workshop*, pages 331–341. Universities of Glasgow and St. Andrews, Departments of Computer Science, Aug 1985. Persistent Programming Research Report 16.

[30] Zongxia Du, Jinpeng Huai, and Yunhao Liu. Ad-UDDI: An active and distributed service registry. In *TES*, pages 58–71, 2005.

[31] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst.*, 19(6):992–1030, 1997.

[32] Eclipse. Eclipse CDO. http://wiki.eclipse.org/CDO, Copyright 2009.

[33] Erki Eessaar. Using metamodeling in order to evaluate data models. In *AIKED'07: Proceedings of the 6th Conference on 6th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases*, pages 181–186, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[34] Mehdi Emadi, Masoud Rahgozar, Adel Ardalan, Alireza Kazerani, and Mohammad Mahdi Ariyan. Approaches and schemes for storing dtd-independent xml data in relational databases. *Trans. on Engineering, Computing and Technology*, 13, May 2006.

[35] ESRI. esri ArcGIS. http://www.esri.com/software/arcgis/index.html, Retrieved April, 2011.

[36] eXo. Java content repository (jcr - jsr 170). http://www.exoplatform.org/portal/public/en/product/oemisv, Retrieved December, 2008.

[37] Sebastian Fischer and Herbert Kuchen. Data-flow testing of declarative programs. In *ICFP*, pages 201–212, 2008.

[38] Daniela Florescu and Donald Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[39] Fornax-Platform. Cartridges. http://fornax.items.de/confluence/display/fornax/Cartridges, Retrieved April, 2011.

[40] Daniel Fotsch and Andreas Speck. Xtc – the xml transformation coordinator for xml document transformation technologies. In *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*, pages 507–511, Washington, DC, USA, 2006. IEEE Computer Society.

[41] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[42] Robert B. France, James M. Bieman, and Betty H. C. Cheng. Repository for model driven development (remodd). In *MoDELS Workshops*, pages 311–317, 2006.

[43] freebXML. OASIS ebXML registry reference implementation project. http://ebxmlrr.sourceforge.net/, July 2007.

[44] Dimitrios Georgakopoulos, Mark F. Hornick, and Amit P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[45] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, 1991.

[46] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[47] Object Management Group. MOF 2.0 / XMI Mapping Specification, v2.1.1. http://www.omg.org/technology/documents/formal/xmi.htm, January 2010.

[48] Xiao guang Zhang. Model driven data service development. In *ICNSC'08*, pages 1668–1673, 2008.

[49] Dirk Habich, Sebastian Richly, Steffen Preissler, Mike Grasselt, Wolfgang Lehner, and Albert Maier. Bpel-dt - data-aware extension of bpel to support data-intensive service applications. In *WEWST*, 2007.

[50] N. Harrison, P. Avgeriou, and U. Zdun. Using patterns to capture architectural decisions. *IEEE Software*, pages 38–45, July/Aug. 2007.

[51] Mary Jean Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM.

[52] S.C Haw and G.S.V. Radha Krishna Rao. Query optimization techniques for xml databases. *International Journal of Information Technology*, 2(1):97–104, 2005.

[53] Carsten Hentrich and Uwe Zdun. Patterns for business object model integration in process-driven and service-oriented architectures. In *PLoP '06: Proceedings of the 2006 conference on Pattern languages of programs*, pages 1–14, New York, NY, USA, 2006. ACM.

[54] Carsten Hentrich and Uwe Zdun. Patterns for process-oriented integration in service-oriented architectures. In *EuroPLoP*, pages 141–198, 2006.

[55] Hibernate. Hibernate. http://www.hibernate.org, Retrieved April, 2011.

[56] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[57] Ibatis. Ibatis. http://www.ibatis.org, Retrieved April, 2011.

[58] IBM. WebSphere Service Registry and Repository. http://www-01.ibm.com/software/integration/wsrr/, Retrieved April, 2011.

[59] IBM. Websphere mq workflow. http://www-01.ibm.com/software/integration/wmqwf/, Retrieved January 2012.

[60] Intalio. Bpm. http://www.intalio.com/bpm, Retrieved January 2012.

[61] S.S. Isloor and T.A. Marsland. The deadlock problem: An overview. *Computer*, 13(9):58–78, 1980.

[62] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEE/IFP Conference on Software Architecture, WICSA*, 2005.

[63] JBoss Community. Jboss messaging. http://www.jboss.org/jbossmessaging, Retrieved January 2012.

[64] JBoss Community. jbpm. http://www.jboss.org/jbpm, Retrieved January 2012.

[65] Rod Johnson. J2ee development frameworks. *IEEE Computer*, 38(1):107–110, 2005.

[66] Matjaz B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition.* Packt Publishing, 2006.

[67] Wolfgang Keller. Mapping objects to tables - a pattern language. In *Proc. Of European Conference on Pattern Languages of Programming Conference (EuroPLOP)1997*, 1997.

[68] Latifur Khan and Yan Rao. A performance evaluation of storing xml data in relational database management systems. In *WIDM '01: Proceedings of the 3rd international workshop on Web information and data management*, pages 31–38, New York, NY, USA, 2001. ACM.

[69] Florian Kiefer, Konstantin Arnold, Michael Künzli, Lorenza Bordoli, and Torsten Schwede. The SWISS-MODEL repository and associated resources. *Nucleic Acids Research*, 37(Database-Issue):387–392, 2009.

[70] Jana Koehler, Rainer Hauser, Jochen Küster, Ksenia Ryndina, Jussi Vanhatalo, and Michael Wahler. The role of visual modeling and model transformations in business-driven development. *Electron. Notes Theor. Comput. Sci.*, 211:5–15, April 2008.

[71] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[72] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Towards a semantic infrastructure supporting model-based tool integration. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 43–46, New York, NY, USA, 2006. ACM.

[73] P. Kruchten, P. Lago, and H. van Vliet. Building up and reasoning about architectural knowledge. In C. Hofmeister, editor, *QoSA 2006 (Vol. LNCS 4214)*, pages 43–58, 2006.

[74] Stefan Kurz, Michael Guppenberger, and Burkhard Freitag. A uml profile for modeling schema mappings. In *ER (Workshops)*, pages 53–62, 2006.

[75] Neil Lang. Schlaer-mellor object-oriented analysis rules. *SIGSOFT Softw. Eng. Notes*, 18(1):54–58, 1993.

[76] Wei Le and Mary Lou Soffa. Refining buffer overflow detection via demand-driven path-sensitive analysis. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 63–68, New York, NY, USA, 2007. ACM.

[77] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE Softw.*, 20(6):35–39, 2003.

[78] Catherine M. Lloyd, James R. Lawson, Peter J. Hunter, and Poul F. Nielsen. The cellML model repository. *Bioinformatics*, 24(18):2122–2123, 2008.

[79] Paul A. Longley, Mike Goodchild, David J. Maguire, and David W. Rhind. *Geographic Information Systems and Science*. John Wiley & Sons; 3rd Revised edition edition, August 2006.

[80] Simone A. Ludwig and S. M. S. Reyhani. Semantic approach to service discovery in a grid environment. *Web Semant.*, 4(1):1–13, 2006.

[81] Xiaogang Ma, Chonglong Wu, Emmanuel John M. Carranza, Ernst M. Schetselaar, Freek D. van der Meer, Gang Liu, Xinqing Wang, and Xialin Zhang. Development of a controlled vocabulary for semantic interoperability of mineral exploration geodata for mining projects. *Comput. Geosci.*, 36:1512–1522, December 2010.

[82] David Masclet. Gisgraphy. http://www.gisgraphy.com/index.htm, Retrieved December, 2010.

[83] Christine Mayr, Uwe Zdun, and Schahram Dustdar. Model-driven integration and management of data access objects in process-driven soas. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet*, ServiceWave '08, pages 62–73, Berlin, Heidelberg, 2008. Springer-Verlag.

[84] Christine Mayr, Uwe Zdun, and Schahram Dustdar. Reusable architectural decision model for model and metadata repositories. In Frank S. Boer, Marcello M. Bonsangue, and Eric Madelaine, editors, *Formal Methods for Components and Objects*, pages 1–20, Berlin, Heidelberg, 2009. Springer-Verlag.

[85] Christine Mayr, Uwe Zdun, and Schahram Dustdar. View-based model-driven architecture for enhancing maintainability of data access services. *Data Knowl. Eng.*, 70(9):794–819, September 2011.

[86] Jose-Norberto Mazón, Juan Trujillo, and Jens Lechtenbörger. Reconciling requirement-driven data warehouses with data sources via multidimensional normal forms. *Data Knowl. Eng.*, 63:725–751, December 2007.

[87] J. Melton and A.R. Simon. *SQL:1999: understanding relational language components*. Morgan Kaufmann series in data management systems. Morgan Kaufmann, 2002.

[88] Jan Mendling, Hajo A. Reijers, and Jorge Cardoso. What makes process models understandable? In *BPM*, pages 48–63, 2007.

[89] Nikola Milanovic, Ralf Kutsche, Timo Baum, Mario Cartsburg, Hatice Elmasgünes, Marco Pohl, and Jürgen Widiker. Model&metamodel, metadata and document repository for software and data integration. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 416–430, Berlin, Heidelberg, 2008. Springer-Verlag.

[90] Jun-Ki Min, Chun-Hee Lee, and Chin-Wan Chung. XTRON: An XML data management system using relational databases. *Information & Software Technology*, 50(5):462–479, 2008.

[91] Krishna B. Misra. *Handbook of Performability Engineering – Quality Engineering and Management*. Springer London, 2008.

[92] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 386–396, Washington, DC, USA, 2009. IEEE Computer Society.

[93] Richi Nayak and Fu Bo Xia. Automatic integration of heterogenous xml-schemas. In *iiWAS*, 2004.

[94] Matthias Nicola and Bert van der Linden. Native xml support in db2 universal database. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1164–1174. VLDB Endowment, 2005.

[95] Hans W. Nissen and Matthias Jarke. Repository support for multi-perspective requirements engineering. *Inf. Syst.*, 24(2):131–158, 1999.

[96] David Nuescheler, Peeter Piegaze, and other members of the JSR 170 expert group. Content Repository API for Java Technology Specification, Java Specification Request 170. http://www.jcp.org/en/jsr/all, May 2005.

[97] Bashar Nuseibeh, Anthony Finkelstein, and Jeff Kramer. Method engineering for multi-perspective software development. *Information & Software Technology*, 38(4):267–274, 1996.

[98] OASIS Web Services Business Process Execution Language (WSBPEL) TC. Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, April 2007.

[99] OASIS/ebXML Registry Technical Committee. Registry Services Specification v2.0. http://www.ebxml.org/specs/ebrs2.pdf, Dec 2001.

[100] Object Management Group (OMG). Business process model and notation (bpmn) version 2.0. http://www.omg.org/spec/BPMN/2.0, Release Date January 2011.

[101] Object Management Group (OMG). Unified modeling language. http://www.uml.org/, Retrieved January, 2010.

[102] Open Geospatial Consortium, Inc. Topic 12: OpenGIS Service Architecture. http://www.opengeospatial.org/standards/as, January 2002.

[103] Open Geospatial Consortium, Inc. OpenGIS Web Feature Service (WFS) Implementation Specification. http://www.opengeospatial.org/standards/wfs, May 2005.

[104] Open Geospatial Consortium, Inc. OpenGIS Catalogue Services Specification. http://www.opengeospatial.org/standards/specifications/catalog, Februrary 2007.

[105] Open Geospatial Consortium, Inc. OpenGIS Geography Markup Language (GML) Encoding Standard. http://www.opengeospatial.org/standards/gml, August 2007.

[106] Oracle Corporation and/or its affiliates. Core J2EE Patterns - Data Access Object. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html, Copyright 2010.

[107] Silke Palkovits and Maria Wimmer. Processes in e-government - a holistic framework for modelling electronic public services. In Roland Traunmüller, editor, *EGOV*, volume 2739 of *Lecture Notes in Computer Science*, pages 213–219. Springer, 2003.

[108] Michael P. Papazoglou. *Web Services: Principles and Technology*. Pearson, Prentice Hall, 2008.

[109] M.P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–65, 2003.

[110] Ken Q. Pu and Xiaohui Yu. Keyword query cleaning. *PVLDB*, 1(1):909–920, 2008.

[111] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2nd edition, 2000.

[112] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *ICSE*, pages 272–278, 1982.

[113] P. Krishna Reddy and Subhash Bhalla. Deadlock prevention in a distributed database system. *SIGMOD Rec.*, 22(3):40–46, 1993.

[114] Luciano Resende. Handling heterogeneous data sources in a soa environment with service data objects (sdo). In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 895–897, New York, NY, USA, 2007. ACM.

[115] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, 1 edition, 2007.

[116] Roberto Riggio, Domenico Ursino, Harald Kühn, and Dimitris Karagiannis. Interoperability in meta-environments: An xmi-based approach. In *CAiSE*, pages 77–89, 2005.

[117] Martin P. Robillard and Frédéric Weigand-Warr. Concernmapper: simple view-based separation of scattered concerns. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 65–69, New York, NY, USA, 2005. ACM.

[118] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In *ER*, pages 353–368, 2005.

[119] Wasim Sadiq and Maria E. Orlowska. Applying graph reduction techniques for identifying structural conflicts in process models. In *In Proceedings of the 11th Conf on Advanced Information Systems Engineering (CAiSE'99*, pages 195–209. Springer-Verlag, 1999.

[120] Claudio Sant'anna, Alessandro Garcia, Christina Chavez, Carlos Lucena, and Arndt v. von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.

[121] Benjamin A. Schmit and Schahram Dustdar. Model-driven development of web service transactions. In *In Proceedings of the Second GI-Workshop XML for Business Process Management,Mar*, page 2005, 2005.

[122] Ali Shaikh Ali, Shalil Majithia, Omer F. Rana, and David W. Walker. Reputation-based semantic service discovery: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(8):817–826, 2006.

[123] Software AG. Webmethods bpms. http://www.softwareag.com/at/products/wm/bpm/default.asp, Retrieved January 2012.

[124] Prawee Sriplakich, Xavier Blanc, and Marie-Pierre Gervais. Supporting transparent model update in distributed case tool integration. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1759–1766, New York, NY, USA, 2006. ACM.

[125] Luke Steller, Shonali Krishnaswamy, and Jan Newmarch. Discovering relevant services in pervasive environments using semantics and context. In *IWUC*, pages 3–12, 2006.

[126] Chuanlong Xia Guangcan Yu Meng Tang. Mapping objects to tables - a pattern language. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009*, December 2009.

[127] The Eclipse Foundation. Eclipse. http://www.eclipse.org/, 2009.

[128] The Eclipse Foundation. Eclipse Modeling Framework Project. http://www.eclipse.org/modeling/emf/, Copyright 2011.

[129] The Eclipse Foundation. Model To Text M2T. http://www.eclipse.org/modeling/m2t/, Copyright 2011.

[130] TIBCO. Bpm. http://www.tibco.com/products/bpm/, Retrieved January 2012.

[131] Albert Tort and Antoni Olivé. An approach to testing conceptual schemas. *Data Knowl. Eng.*, 69:598–618, June 2010.

[132] Huy Tran, Uwe Zdun, and Schahram Dustdar. View-based and model-driven approach for reducing the development complexity in process-driven SOA. In Witold Abramowicz and Leszek A. Maciaszek, editors, *Business Process and Services Computing: 1st International Conference on Business Process and Services Computing (BPSC'07), September 25-26, 2007, Leipzig, Germany*, volume 116 of *LNI*, pages 105–124. GI, 2007.

[133] Mark Turner, David Budgen, and Pearl Brereton. Turning software into a service. *Computer*, 36:38–44, 2003.

[134] J. Tyree and A. Ackerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(19–27), 2005.

[135] Lorenzino Vaccari, Pavel Shvaiko, and Maurizio Marchese. A geo-service semantic integration in spatial data infrastructures. *International Journal of Spatial Data Infrastructures Research*, 4:24–51, 2009.

[136] Sekhar Vajjhala and Joe Fialli. The Java Architecture for XML Binding (JAXB) 2.0. http://jcp.org/aboutJava/communityprocess/final/jsr222/index.html, April 2006.

[137] Reind P. van de Riet. Twenty-five years of mokum: For 25 years of data and knowledge engineering: Correctness by design in relation to mde and correct protocols in cyberspace. *Data Knowl. Eng.*, 67(2):293–329, 2008.

[138] Vânia Maria Ponte Vidal, Fernando Cordeiro Lemos, and Fábio Feitosa. Translating wfs query to sql/xml query. In *GeoInfo*, pages 174–190, 2005.

[139] Markus Völter and Thomas Stahl. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley, 2006.

[140] W3C. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2001.

[141] Jun Wang, AiRong Yu, XiaoYi Zhang, and Lei Qu. A dynamic data integration model based on soa. In *ISECS International Colloquium on Computing, Communication, Control, and Management*, pages 196–199, Washington, DC, USA, 2009. IEEE Computer Society.

[142] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures.* Springer, 2007.

[143] U. Zdun and S. Dustdar. Model-driven and pattern-based integration of process-driven soa models. In *Int. J. Business Process Integration and Management*, volume 2 Number 2, pages 109–119, 2007.

[144] Guanqun Zhang, Xianghua Fu, Shenli Song, Ming Zhu, and Ming Zhang. Process driven data access component generation. In *DEECS*, pages 81–89, 2006.

[145] Ye Zhou and Edward A. Lee. A causality interface for deadlock analysis in dataflow. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 44–52, New York, NY, USA, 2006. ACM.

[146] Fujun Zhu, Mark Turner, Ioannis Kotsiopoulos, Keith Bennett, Michelle Russell, David Budgen, Pearl Brereton, John Keane, Paul Layzell, Michael Rigby, and Jie Xu. Dynamic data integration using web services. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, page 262, Washington, DC, USA, 2004. IEEE Computer Society.

[147] O. Zimmermann, T. Gschwind, J. Kuester, F. Leymann, and N. Schuster. Reusable architectural decision models for enterprise application development. In S. Overhage and C. Szyperski, editors, *Quality of Software Architecture (QoSA) 2007*, Lecture Notes in Computer Science, Boston, USA, July 2007. Springer-Verlag Berlin Heidelberg.

[148] Olaf Zimmermann, Uwe Zdun, Thomas Gschwind, and Frank Leymann. Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 157–166, Washington, DC, USA, 2008.

Christine Mayr received her master degree in computer science from the Vienna University of Technology in 2002. She has more than 10 years of working experience in the field of software analysis, design, development and management. Currently she is working as an IT-analyst in the Federal Computing Centre of Austria. One major part of her work is data modeling and database design for service-based applications. In addition, she is an external Ph.D. student at the Distributed Systems Group of Vienna University of Technology. Her research interests are model-driven development, service-oriented computing, and architectural decisions.

Personal Information

Born: 16th May 1979 in Nuremberg, Germany
Citizenship: Austrian

Education

- **10/ 2007– 6/ 2012: PhD Studies in Computer Science**, Vienna University of Technology, Austria
- **10/ 1997– 6/ 2002: Studies in Computer Science**, Vienna University of Technology (graduation with distinction), Austria

Jobs

- **08/ 2006 – now: IT– Analyst**, Federal Computing Centre of Austria, Vienna
- **07/ 2001 – 7/ 2006: IT– Consultant**, Wincor Nixdorf GmbH, Vienna, Austria
- **10/ 2000 – 01/2001: Tutor**, Vienna University of Technology, Institute of Computer–Aided Automation for the course "Introduction into programming"
- **10/ 2000 – 07/ 2001: Working Student (part– time)**, Wincor Nixdorf GmbH, Vienna, Austria
- **07/ 2000 – 09/ 2000: Working Student**, Wincor Nixdorf GmbH, Vienna, Austria
- **08/ 1999 – 09/ 1999: Working Student**, Department for Information and Communication, Siemens AG Munich, Germany