

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

Christine Mayr^a, Uwe Zdun^b, Schahram Dustdar^a

^a*Distributed Systems Group, Information Systems Institute, Vienna University of Technology, Austria*

^b*Faculty of Computer Science, Research Group Software Architecture, University of Vienna, Austria*

Abstract

In modern service-oriented architectures, database access is done by a special type of services, the so-called data access services (DAS). Though, particularly in data-intensive applications, using and developing DAS is very common today, the link between the DAS and their implementation, e.g. a layer of data access objects (DAOs) encapsulating the database queries, still is not sufficiently elaborated, yet. As a result, as the number of DAS grows, finding the desired DAS for reuse and/or associated documentation can become an impossible task. In this paper we focus on bridging this 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 of different stakeholders and are scalable with all types of DAS implementations. In this paper we show that 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.

1. Introduction

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 specified by using a web service description language (WSDL) [60]. Besides invoking services, process activities can perform human tasks, do transformations and/or invoke other process activities. Service repositories [25, 12] can be used to manage services and support service discovery at runtime. As shown in Figure 1, the process activity queries a service repository in order to find a suitable service for dynamic invocation (1). Typically, services need to read or write data from a database. Nowadays, this data access is done by so-called data access services

Email addresses: christine.mayr@inode.at (Christine Mayr), uwe.zdun@univie.ac.at (Uwe Zdun), dustdar@infosys.tuwien.ac.at (Schahram Dustdar)

(DAS). DAS are variations of the ordinary service concept: They are more data-intensive and are designed to expose data as a service [56]. They can either be invoked by another service or by a process activity directly. As depicted in Figure 1, a service repository returns a service, that is running on a DAS provider. Eventually, the process activity dynamically invokes the service on a certain DAS provider (2).

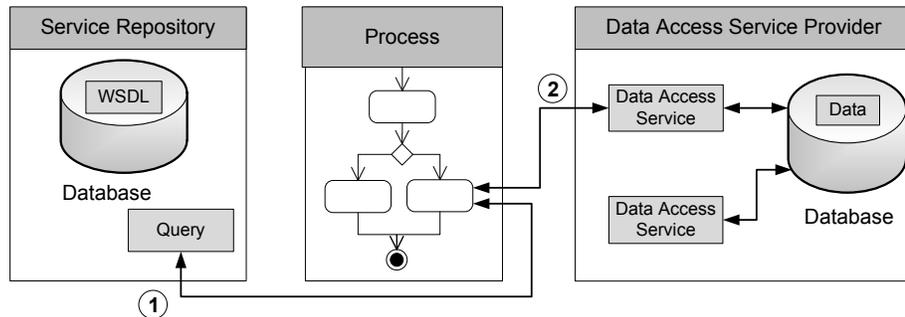


Figure 1: Data Access in a process-driven SOA

In object-oriented environments, DAS commonly use a layer of data access objects (DAOs) to read and write data from a relational database management system (RDBMS). According to the JEE pattern catalog [38], the DAO pattern abstracts and encapsulates all access to the data source and provides an interface independent of the underlying database technology. The DAO manages the connection with the data source to obtain and store data.

Status quo. A process-driven SOA is an architectural style for developing large business applications. Accordingly, a huge number of processes, processes activities, services, and in particular data access services need to be managed. Nowadays, business process execution languages such as [41] are used as the missing link to assemble and integrate services into a business process [27]. These business process execution languages provide higher level control for services as they describe the services to be invoked and which operations should be called in what sequence. In order to maintain and integrate processes and services, much research work has been done. However, these business process languages do not integrate the semantics of an invoked service such as which DAS reads or writes which data. In contrast, they rather regard the process internal data read and written by process activities.

Basic problem. Unfortunately, the relationships between the DAS, the underlying DAOs, and the data storage schemes are not sufficiently explored, yet. Figure 2 overviews these missing links: The DAS in the service repository are neither associated with the DAS source code, nor with the service internal documentation, nor with the data storage schemes. Accordingly, the service internal documentation in the middle of the figure is loosely coupled with the DAS, the DAS source code and the data source schemes. However, from our experiences, in order to efficiently maintain DAS, a further integration of the DAS, the DAS source code, DAS documentation, and the data storage schemes is compulsory. In the following we describe the

related problems experienced when maintaining, reusing, documenting, tracing and developing data access in a large enterprise in more detail.

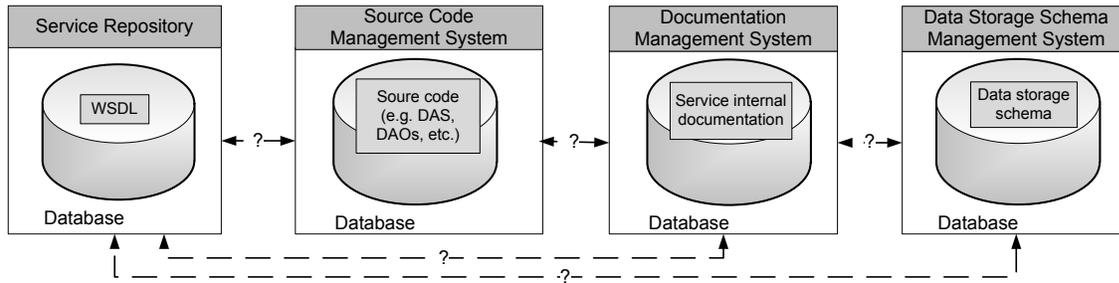


Figure 2: Missing Links between Data Access Services, Source Code, Documentation and Data Storage Schemes

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 this change. If a database table schema is redesigned e.g. in case of altering a column, it is 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 may be obsolete and should not be available anymore. Due to lacking integration of DAS and the data storages, further elaboration to improve maintainability of DAS is required.

Lack of DAS documentation and traceability. Moreover, when maintaining processes in a process-driven SOA, developers need to understand the relationships between process activities, the invoked DAS, the DAOs, and the used data storages. In order to relate DAOs to data storages, they need the information which database tables relate to which data objects and which data objects are used by a DAO. 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 [28].

Insufficient reuse of data access best practices. DAO implementations use techniques for mapping data objects to relational databases. 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. Up-to now attaining good performance still requires careful optimization based on expert knowledge, which can make programs difficult to maintain and evolve [13]. 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, data analysts require mainly information about which DAOs access which data, service developers require DAOs rather as interfaces to the data, and software architects require the big picture of service/DAO interconnection

Our approach. In smaller environments, development techniques like naming conventions and documentations may be also be useful to handle the problems above. However, in larger-scale environments, when the number of software components grows, more sophisticated methods to manage traceability and maintainability are necessary [32]. In this paper we attack these problems by presenting our view-based model-driven data access architecture (VMDA) built on the following basic concepts.

Four basic concepts. In order 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. In addition, as object oriented programming (OOP) is typically used to implement services, we focus on DAO integration into the DAS. However our architecture approach can be reused to integrate other types of data access implementations into DAS. We use the **model-driven development (MDD)** [59] approach to be able to abstract DAS from a higher level than the source code layer. Accordingly, our DAS have highly structured DAO implementations that can be used as the basis for documenting the relationships between DAS, the DAOs and the data storages. Another useful development aspects of MDD, we can make use of, are automatic source code generation and deployment. As service repositories such as WSRR [25] and UDDI [12] provide management support for services, we introduce a **data access service (DAS) repository** storing DAS models and model instances. More precisely, our DAS repository stores models and model instances of the DAS, the DAOs, the ORMs, the data objects and the data storage schemes including the relationships between them. The DAS repository provides services to manage DAS/ DAOs, in particular a query service to request DAS and/or DAOs from the DAS repository by diverse search criteria. As a result, developers can query all DAS and DAOs respectively belonging to a certain data storage schema. The other way around, all database storage schemes can be retrieved that are used by a given DAS or DAO. In this paper, we also prove, that our DAS repository offers fast and efficient retrieval of DAS, DAOs and data storage schemes. Our model-driven solution for better maintaining DAS in process-driven SOAs is based on the **view-based modeling framework (VbMF)** introduced in our earlier work [55]. This framework aims at separating different concerns in a business process into different **views**. The main idea in our VbMF approach is to enable stakeholders to understand each view on its own, without having to look at other concerns, and thereby reduce the development complexity. The data-related extension of VbMF, the **view-based data modeling framework (VbDMF)**, introduces a layered data model for accessing data in process-driven SOAs [32]. The concept of separation of concerns contributes to increase maintainability of processes, services, the DAOs and the underlying data storage schemes in process-driven SOAs.

Summary. To sum-up, our novel contribution combines four basic concepts (DAS, MDD, DAS repository, VbMF/ VbDMF) in order to solve the problems and integration gaps described above. Basically our model-driven architecture aims at improving software development productivity and maintainability by enhancing traceability, documentation and reuse of DAS. Moreover, our approach opens a wide range of applications in order to improve maintainability. As an example, a database query analyzer could be integrated into our model driven data access architecture. The DAS repository could be fed with these query benchmarks in order to evaluate the DAS and thus improve data access reuse. Furthermore, we have developed an adequate tool support based on our architectural view concept leading to a highly inter-operable system.

This paper is organized as follows: First, Section 2 describes the related work and discusses how our work distinguishes from the related work. Next, in Section 3 we give a basic overview of VbMF and VbDMF. In the following Section 4 we present our view-based model-driven data access architecture (VMDA). Section 5 looks deeper into the DAS repository, describes the basic architectural decisions, the underlying DAS repository services, and the data model. Section 6 shows our prototype tooling and hence describes our VMDA from the user's point of view. In Section 7, we illustrate the applicability of our approach by an industrial case study in the area of modeling jurisdictional provisions in the context of a district court. Section 8 underlays our approach contributions with quantitative evidences. Finally, Section 9 concludes and outlines future activities.

2. Related Work

In this section we present related work from existing literature, standards, and known uses. We also emphasize the contribution of our work by explaining how our work compares to these related work. As our approach is composed of four basic concepts (DAS, repository, views, MDD), we try to compare with representatives of each field. Finally, Table 1 summarize this comparison.

2.1. DAS Architecture Approaches

The most related to our work is probably the architectural approach of Zhu et al. [61]. 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, 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 a SOA environment [49]. He describes how to efficiently handle data access within a service data objects (SDO) [49] environment.

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) [48] to transform XML data formats into objects of object-oriented programming languages. In our approach, for example, these XML-to-Object transformations are encapsulated by the DAS used for uniformly accessing heterogeneous data sources.

There are several approaches for semantic knowledge discovery e.g. [53],[10]. Representatively, we refer to the software architecture of Cannataro et al. [10] for distributed knowledge discovery. The paper discusses how the Knowledge Grid can be used to implement distributed data mining services. The services are responsible for the search, selection, extraction, transformation and delivery (data extraction services) of data to be mined. On the basis of the user requirements and constraints, the services automate the searching and finding of data sources to be analyzed by the data mining tools. 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 [30]. In this paper, 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 free text by semantic services. Moreover, we can reuse the high-structured DAS for model-to-code and model-to-documentation transformations.

2.2. *Service Repositories*

Our work is inspired by current web service registry standards such as UDDI [12], ebXML [42], WSRR [25], and WSIL [4]. Like our approach, EbXML web service registries [42] have interfaces that enable submission, query, and retrieval of the contents of the registry. Standards such as UDDI have enabled service providers and requesters to publish and find services of interest through UDDI Business Registries (UBRs), respectively. However, UBRs are not adequate enough for enabling clients to effectively find relevant web services due to a variety of reasons [2]. E.g. due to missing key words and unsatisfactory documentation retrieving DAS is often impossible. Consequently, there is a need to utilize requester and service context during the discovery process [54]. 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 a process-driven SOA, we adopted the basic CRUD interface abstractions, used in these approaches, and integrated them into our DAS repository architecture. Moreover, these service repositories such as UDDI [12] strictly separate the interfaces from their implementation. In contrast to these approaches, our VMDA integrates the DAS and their implementation, 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.

2.3. Model Repositories

There are many model repositories that store meta models, models and model instances such as [34] and [39]. An interesting approach is the one of Milanovic et al. [34] who present the design and implementation of a repository that supports storing and managing of various artifacts such as meta-models, models, constraints, meta-data, specifications, etc. They illustrate the repository's data model specifying the stored artifacts and artifact meta-data. Furthermore, they give an overview of the repository architecture, and describe how to manage artifacts from the repository point of view. However, they do neither specify client-server interactions nor how to synchronize with other repositories. In our work, we also describe the basic repository services from a user's point of view.

Nissen and Jarke's encouraging work propose repository support for goal-oriented inconsistency management in customizable multi-perspective modeling environments [39]. 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 these 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 sub views 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 while DAO developers can focus on describing DAOs of the DAO view. In contrast, Nissen and Jarke's enable the representation of conflicting perspectives, more precisely of different modeling levels, by adding a separation mechanism called modules to the formal conceptual modeling language Telos. Consequently, they only present requirement-level specific views for stakeholders focusing on a concern with a certain modeling levels. Besides supporting modeling-level specific views, our views can even be domain-specifically tailored to the stakeholders requirements.

Min et al. [35] present an XML data management system using a relational database as a repository that translates a comprehensive subset of 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 often based on a meta-model. Hence, if the models contain embedded meta-data elements, then using XQuery to query the model elements, might 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 views model instances and view models by different search criteria. Thus, our simple search language supports user-friendly key word search in XML models whereas Min et. al. 's approach lacks usability, when stakeholders only have a limited knowledge of the meta-data within XML models.

There are several implementations of model repositories [37],[16]. An example of a model repository implementation is the Netbeans MetaData Repository (MDR) [37] to store model and model instance data. As our approach, MDR supports storing and managing of different modeling levels in the repository. As opposed to MDR, we support structured searching mechanisms in order to improve maintainability of the model instances in the software development process.

2.4. 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. [50] 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 are can only be retrieved, when the relevant source code section have been mapped to concerns, before. Due to our model-driven approach, we can retrieve all model elements and their relationships by diverse search criteria. Thus our view-based model instances can be used as the basis for model-to-documentation transformations.

Nuseibeh et al. [40] examined method engineering in the context of multi-perspective software development, as exemplified by the ViewPoints framework. In order to manage the diversity in composite systems they define ViewPoints, like our views, to be loosely coupled, locally managed, and distributable objects that encapsulate representation knowledge, development process knowledge and specification knowledge about a system and its domain. Thus a single ViewPoint contains a partial specification described in a formal notation and developed by following a particular development strategy. In contrast to our approach, their view-based solution is an organizational framework. Moreover, our DAS repository is 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.

2.5. Model-driven frameworks and other developmental related work

Nowadays, there is still a missing link between the programming components and the data storage schemes [32]. This gap results in several object-relational mapping (ORM) problems [21]. Accordingly, up-to now

attaining good performance still requires careful optimization based on expert knowledge, which can make programs difficult to maintain and evolve [13]. There are two common model-driven frameworks, so called cartridges that, as our approach, can close the gap between the programming components and the data storage schemes. In this paragraph we developmentally compare the models, because to the best of our knowledge there is no literature describing model-driven solutions in order to bridge the gap between DAS, DAOs and database queries. As our DAS/DAO models are based on our VbDMF [55], data access related models in other model-driven frameworks, such as AndroMDA's EJB3 cartridge [3] and the Fornax platform [20], are also related to our approach. AndroMDA's EJB3 cartridge [3] and the Fornax platform [20] respectively generate a persistent tier from the DAO models. Each DAO model specifies the DAO operations encapsulating the database queries. These UML-based modeling frameworks provide elaborated models for specifying DAO operations with Hibernate [22]. However, they do not focus on supporting different ORM frameworks nor on integrating different-level views tailored to the requirements of certain stakeholders.

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 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 times.

3. The View-based Data Modeling Framework

In this section, we present our view-based data modeling framework [32] used to model the views managed by our view-based model-driven data access architecture (VMDA).

3.1. 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. When developing and maintaining data access services (DAS), stakeholders are interested in different concerns. System architects typically focus on the system's runtime configuration and thus concentrate on the business process execution endpoint of the process flow e.g. described by BPEL [41], the service endpoints, and the database connections. On the contrary, business process developers need an overview of the elements of a process flow such as the process activities. As process activities can invoke DAS, business process developers, like the DAS developers, are also interested in basic DAS descriptions elements such as DAS operations, endpoints of

Table 1: Comparison of Related Work

Representative, Short Description/Requirement	Zhu et al. [61], DAS architecture	Resende [49], Service Data Objects (SDO)	Camataro et al. [10], Semantic Knowledge Discovery	UDDI [12], Service Repositories	Milanoic et al. Model Repository	Nissen and Jarke [39], Multi-perspective modeling and environments	Min [35], XTRON: An XML data management system using relational databases	Robillard et al. [50], Concern Extraction	Nuseibeh et al. [40], Multi-perspective software development	Our approach, View-based model-driven data access architecture
Supports Data Access as a service	yes. The integration broker service integrates different data access services and other functional services in order to provide the end-user with an integrated uniform view of the data.	yes. The DAS component is responsible to provide access to data sources.	yes. The knowledge directory service is responsible for maintaining a description of all the data and tools used in the Knowledge Grid.	yes. A UDDI registry must have at least one node that offers a Web service compliant Inquiry/API set.	yes, by the remote access layer of the BIZCYCLE Repository Architecture	not subject of this work	not subject of this work	not subject of this work	not subject of this work	yes, the DAS are specified by VMDM, stored in the DAS repository and managed by our VMDA
Supports management (CRUD) of artifacts	yes	yes	yes	yes	yes	yes	yes	not subject of this work	not subject of this work	yes
Supports structured search of artifacts	yes	no	yes	no	not specified	not specified	yes, by the standard XQuery language	not subject of this work	not subject of this work	yes, by our own proprietary language
Supports different views	not subject of this work	not subject of this work	not subject of this work	not subject of this work	not subject of this work	no	no	yes, the Concern Mapper extracts concerns from the source code	yes, the viewpoint concept enables stakeholders to concentrate on different perspectives	yes, our view-based approach is based on the concept of separation of concerns. Different views are tailored to the requirements of certain stakeholders.
Supports different modeling levels	no	no	no	no	no	yes, due to separation of multiple perspectives (meta-model level, conceptual modeling level, model level, instance level)	no	no	no	yes, our VMDA supports storing models and model instances.
Tool Support	yes, they developed two major prototypes	no	no	There are several UDDI implementations e.g. [12]	yes, the BIZCYCLE repository has been prototypically implemented using J2EE technology and JBOSS as the application server.	no	yes, by performing XQuery in a GUI	Tool support by Concern Mapper, an Eclipse plug-in for experimenting with techniques for advanced separation of concerns.	The Viewer is a prototype environment supporting the View-Points framework developed by the Distributed Software Engineering Group at Imperial College.	Provided by the view-based repository client

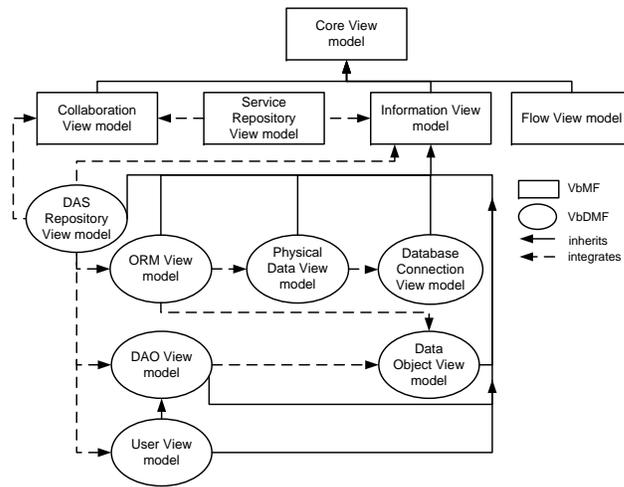


Figure 3: VbMF and VbDMF

running services, the messages, and the data types. When implementing DAS, the DAS developers also need to view descriptions of the underlying DAO operations in order to invoke them. In addition, they need to know available database connections in order to test the DAS. In contrast, DAO developers focus more on the DAO operations with the objection relational mappings (ORMs) between the database tables and the data objects. For this, they also need an overview of the data storage schemes. In order to test the DAOs, like the DAS developers, the DAO developers need to know available database connections. Database administrators are primarily interested in the database connections of a DAS. Finally, database designers typically only need a view of the data storage schemes in order to describe the data model. The concept of separation of concerns aims at reducing this complexity by enabling stakeholders to focus on their own concerns. In our approach, we apply the concept of separation of concerns and use a view-based modeling framework specifying views tailored to the requirements of different stakeholders. Accordingly, stakeholders can concentrate on their own concerns without keeping themselves busy with unnecessary details [55].

3.2. Basic Overview of the View-based Modeling Framework

The view-based data modeling framework (VbDMF) is an extension of the view-based modeling framework (VbMF). VbDMF contains the basic views, for modeling DAS in a process-driven SOA. Figure 3 illustrates the relationships between VbMF and VbDMF: The rectangles in Figure 3 display the basic models of VbMF whereas the ellipses denote the data-related models of VbDMF. In VbMF and VbDMF new architectural views can be *designed*, existing models can be *extended* by adding new features, views can be *integrated* in order to produce a richer view, and using *transformations* platform-specific code or documentation can be generated. VbMF basically consists of the following views:

- The Core View model is the basic VbMF model and is derived from the Ecore meta-model [15].

- 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.
- The Service Repository View model specifies a combined view by integrating the Collaboration View and the Information View.

As displayed by the dashed lines in Figure 3 the view models of VbDMF extend the Information View model of VbMF. The dotted lines in the figure are used to display view integration that is used by the DAS Repository View to integrate the Collaboration View, the Information View, the ORM View, the User View, and the Data Object View to produce a combined view.

3.3. View-based Data Modeling Framework (VbDMF)

In the following, we specify the VbDMF models, shown in Figure 3, in more detail. Please note, that, in this paper, 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 the paper. Moreover, as our VMDA is intended for use in larger environments, we propose ORM instead of the more primitive Java Database Connection (JDBC) interface to access the data.

Database (DB) Connection View model. The Database Connection View model describes the database connections, each comprising a list of user-defined connection properties. Typically, they are the system architects and database administrators who need an overview about the database connections.

Physical Data View model. This view model is primarily intended for database designers, and DAO developers who rely on detailed physical data design. The Physical Data View integrates the Database Connection View.

Data Object View model. In object-oriented programming languages, information is stored in the objects' member variables. We provide a conceptual, technology-independent model integrated by the ORM View model and the DAO View model. The Data Object View is typically used by the DAO developers.

Object Relational Mapping View model. The Object Relational Mapping (ORM) View model is a technology-dependent model that provides the basis for specifying object relational mapping mechanisms in VbDMF typically specified by the DAO developers. In order to support special features of ORM frameworks such as Hibernate [22] and Ibatis [24], developers should specify a new technology-dependent

model by model extension. The basic model specifies a mapping between the Data Object View and the Physical Data View.

DAO View model. The DAO View model basically specifies the DAO operations to read and write from a data storage. The view integrates the Data Object View and is typically used by the DAO developers.

User View model. The User View model gives an overview of the registered and published DAS, and the users who registered and published the DAS.

DAS Repository View model. The DAS Repository View model is a combined view model. The DAS Repository View integrates the Collaboration View, the Information View, the Object Relational Mapping (ORM) View, the Data Object View and the User View. As a result of this view integration, a DAS repository service can process arbitrary queries for retrieving a specific DAS operation or an underlying DAO operation by joining elements from different views.

To summarize, VbMF and VbDMF focus on reducing the development complexity in process-driven SOAs. By exploiting the concept of separation of concerns, we enable stakeholders to concentrate on tailored views. By the mechanism of view integration, it is possible to combine these tailored views. In this way, we can, in particular, connect the DAS, DAOs and data storage schemes.

4. Architecture Overview

In this section we present the big picture of our view-based model-driven data access architecture (VMDA). As already mentioned in Section 1, 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 VbMF/ VbDMF 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

In the following we explain the VMDA components depicted in Figure 4: There are many solutions of use based on model repositories to efficiently manage structured elements [52], [8]. In this paper, we also take advantage of a model repository to manage both the DAS/DAO view models *specifying the view model instances* and the view model instances describing the *DAS* and *DAOs*. By the way, we use the term *view* to refer to view model instances throughout the paper. Our **data access service (DAS) repository** is the main part of our VMDA, as it is used to manage the view model instances and view models of the DAS, DAOs, the underlying data storage schema, and the relationships between them.

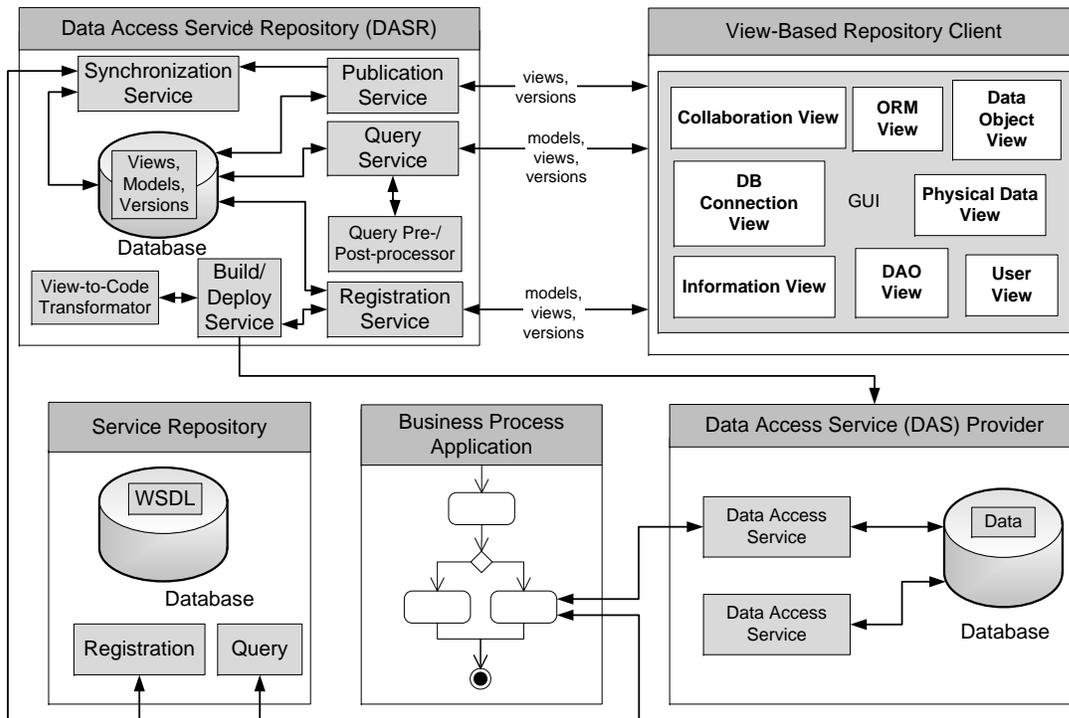


Figure 4: View-based Model-driven Data access Architecture (VMDA)

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. Our query service uses a **query pre-/ post-processor** to perform query transformations to translate proprietary query languages into valid database queries. In Section 6.2.1 we define our own proprietary query language that is used by our prototype implementation to query view models and views by different search criteria. Via the query service both view models and view model instances can be retrieved.

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 openArchitectureWare's (oAW) [47] Xpand language. After source code generation, the DAS can be automatically built and deployed on a certain **DAS provider**. The DAS provider information can be optionally set by the 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. Hereto, the publication service invokes the **synchronization service** that publishes the DAS to the other repositories. The synchronization service handles synchronization problems arising when data synchronized to the other repository does not match the data currently at the constituent repository [1]. This may occur, for instance, if a DAS endpoint, stored redundantly in both the DAS repository and in a **service repository**, changes. If so, there is a need to replicate the new DAS endpoint from the service repository to the DAS repository or inversely. The synchronization service synchronizes 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 DAS 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 [7].

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 paper we do not focus on dynamic invocation of services, however, in order to describe our approach we keep the whole SOA in view. Moreover, as the tools are what gives value to a repository [6], we use a **view-based repository client** based on VbMF and VbDMF. The view-based repository client exploits the concept of separation of concerns and hereby improves maintainability of the data accesses in process-driven SOAs [51]. We present our view-based repository client implementation in Section 6.

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.

5. The Data Access Service (DAS) Repository

As the DAS repository is the central component of our VMDA, we go deeper into the basic architectural decisions that have to be made while designing a DAS repository. Secondly, we present the services for querying, registering, publishing and synchronizing the DAS repository artifacts in more detail. Finally, we focus on the entities of the repository's view-based data model.

5.1. Basic Architectural Decisions

5.1.1. Stored Artifacts

During the design of a DAS repository, a variety of architectural decisions have to be made. A basic architectural decision is whether to store view models and/or view model instances in the DAS repository. In order to quantify the number of view models and view model instances in the DAS repository, as shown in

Figure 3, 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 100 DAS, 1200 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 1 % of the views.

We decided to store both the view model instances and the view models for the following reasons: Storing the view models in the repository enables important development aspects such as validating the view model instances. We store the view model instances because they contain the basic elements that can be, in particular, applied to generate the runnable DAS.

5.1.2. Data Redundancy and Synchronization

Another decision is whether the view model instances, logically belonging to other repositories, shall be redundantly stored in both the owner repository and in the DAS repository, or solely in the owner repository. For instance, the web service description language (WSDL) is usually owned by a service repository. If the view model instances (basically the Collaboration and the Information View), specifying a WSDL, are registered to the DAS repository, there are two possibilities in regard to storing these view model instances: The DAS repository can either store the view model instances redundantly in the DAS repository and delegate the request to the service repository, or the DAS repository can only delegate the request to the service repository without storing the view model instances in the DAS repository. In both cases, as illustrated in Figure 4, before delegating the registration request to the service repository, the DAS repository registration service has to transform the views according to the service specification of the service repository. Unfortunately, the service repository specification does not support structured elements. Accordingly, without structured elements, we cannot support structured queries of view elements, which is a main contribution of our architecture concept. 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. As already mentioned in Section 4, 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. Below, in Section 5.2.5 we specify the synchronization service used to synchronize the data between the DAS repository and the other related repositories.

5.1.3. Version management

When designing a repository, architects have to decide whether to add version information or not [33]. We support version management for view models and view model instances stored in the DAS repository. Like source code in a source code version management system has to comply with a certain version of programming language, our view model instances have to comply with certain model versions.

With our approach, if a model lacks some features, stakeholders can register and publish a revised version of

this model to the DAS repository. Then, a lot of other developers can reuse this model, and develop model instances complying with this new model.

During the publication process, a view model instance version is transferred to other repositories that are connected with the DAS repository. In case of the service repository, we use the UDDI tModel structure to store the versioning information. The service repository model contains a structure known as tModelInstanceInfo which in turn contains instanceDetails. The instanceDetails data structure is extensible and allows us to add version number for the service being registered. Adding a version number to instance details enables tModels to communicate service versioning information along with other information used to describe the service [19]. Furthermore we support change log meta-data about which user inserted or updated a certain repository model and view alternatively [33].

By the way, we do not manage different versions of the Ecore meta-model [17] that is used to specify the view models, as mentioned before. If a new Ecore meta-model version shall be used, all view models and view model instances have to be upgraded to comply with this new Ecore meta-model version.

5.1.4. Version compatibility

In contrast to managing view model instances of the same model version, managing view model instances belonging to different model versions comes along with consistency problems. If a model element of a new model version has been deleted or renamed, the query service need to support different different view model versions. As a result, querying view model instances can become very complex. In addition, if a model element of a new view model has been renamed, search results can become worse, because two different model elements of the older model version and the new model version, respectively, can contain the same values.

Thus, we decide that new versions of models have to be downward compatible with previous versions. Accordingly, we permit new view model elements in order to being able to improve view model definitions to be able to search for these new view models and view model instances. However we do not allow renaming and deleting view model elements, because this results in a complex query service and worse search results. Although, this downward compatibility comes along with a limited flexibility to define new models. For instance, if a certain element is not necessary anymore, it must not be deleted. Otherwise the new view model version would not have been downward compatible with the previous versions. Hence, we recommend to regularly upgrade view model instances to comply with the newest view models if the view model version changes. Accordingly, as soon as all view model instances are upgraded to the newest model version, it is no more necessary for the view models to be downward compatible with older view model versions.

5.2. Services

In the following we explain the DAS repository services in more detail. Please note that all services support both view models and view model instances.

5.2.1. 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 need 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 storages 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 meta-data 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, meta-data artifacts such as affiliation and version information can be used to enable better search results [33]. Furthermore, all entities of the VbDMF model can be used as search criteria. Thus the query service is flexibly extensible for view model changes. In Section 6.2 we present our lightweight query language used by our prototype implementation in order to query view models and view model instances.

5.2.2. 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 5(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.

5.2.3. Build/ Deploy Service

By using the build/ deploy service, DAS can be deployed on a DAS provider. As shown in Figure 4, 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

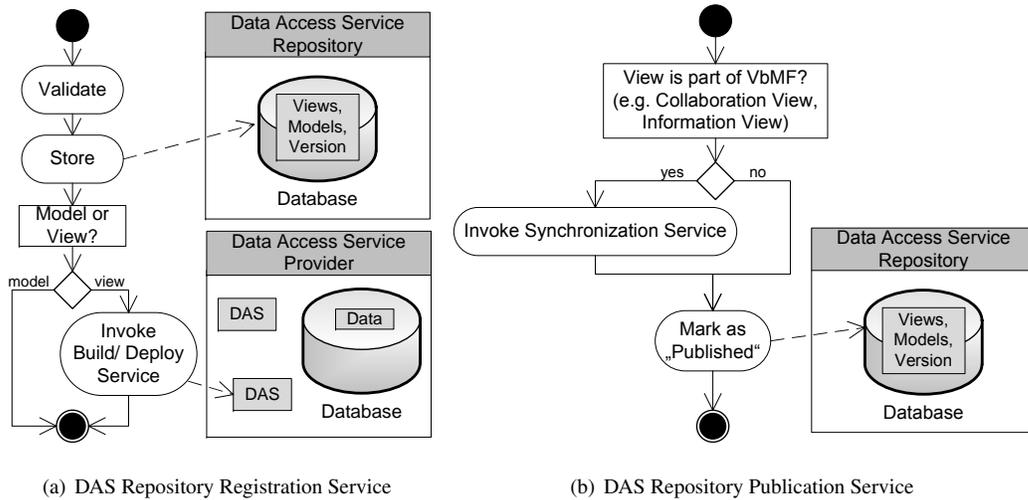


Figure 5: Registration Service and Publication Service

successfully testing the DAS, they should publish the DAS to other users and repositories via the following publication service.

5.2.4. 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 5(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 is registered to the service repository. In this case, the publication request is delegated to the synchronization service. Likewise, data-related views can be synchronized to a schema repository [7]. 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.

5.2.5. Synchronization Service

As we store data redundantly in both the DAS repository and other repositories such as the service repository, we have to deal with synchronization problems. The synchronization service is used to replicate view model instances between the DAS repository and other repositories. Hereto, new view model instances have to be replicated from the DAS repository to the other repositories. Likewise, related data from the other repositories need to be transferred to the DAS repository.

On one hand, frequent data replications have to be done to synchronize the repositories, but it is 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]. Therefore, our DAS repository synchronization service acts as a Push Model Data provider to directly push publication requests from the DAS publication service to other repositories such as to service repository side. In example, when the DAS endpoint in the Collaboration View has been adapted, the adapted Collaboration View is published to the DAS repository, and finally the new DAS endpoint need to be replicated to the service repository. In contrast to these service repositories such as UDDI [12], that can not delegate incoming registration requests to other repositories [61], the DAS repository synchronization service can actively replicate DAS repository data to other repositories. Hereto, the DAS repository synchronization service transforms the view model instances to comply with the Application Programming Interface (API) of the other repository services.

In literature several approaches for repositories to implement better synchronization [1],[14] are proposed. In all these solutions, best estimation and syndication can be reached with the pull based model [1]. The problem is, that the UDDI [12] service repository, such as many other repositories, implement no synchronization model, neither the pull-based nor the push-based synchronization model. Thus, our DAS repository uses active monitoring mechanisms to transfer changes from other repositories such as from UDDI to the DAS repository. By these active monitoring mechanisms, the DAS repository synchronization service may find the latest information of UDDI transparently and conveniently [14]. During this periodic monitoring, the synchronization service checks, if there exist newer data or newer versions of data in the UDDI. Afterwards, the new UDDI data is re-engineered into the DAS repository.

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 6. Moreover, in our Case Study Section 7 we apply the DAS repository services using concrete use cases.

5.3. The DAS Repository View Model

In Section 3, we have given a basic overview of VbMF and VbDMF. In this section we present our DAS Repository View model with a decisive goal in the context of this paper: 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 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 paper, we use DAOs as exemplary DAS implementation for object-oriented environments.

In Figure 6, we display the relationships between the model entities and their related VbMF and VbDMF views as a UML diagram. The `Database` 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. A `Database` consists of zero or more `Tables` that in turn holds a list of

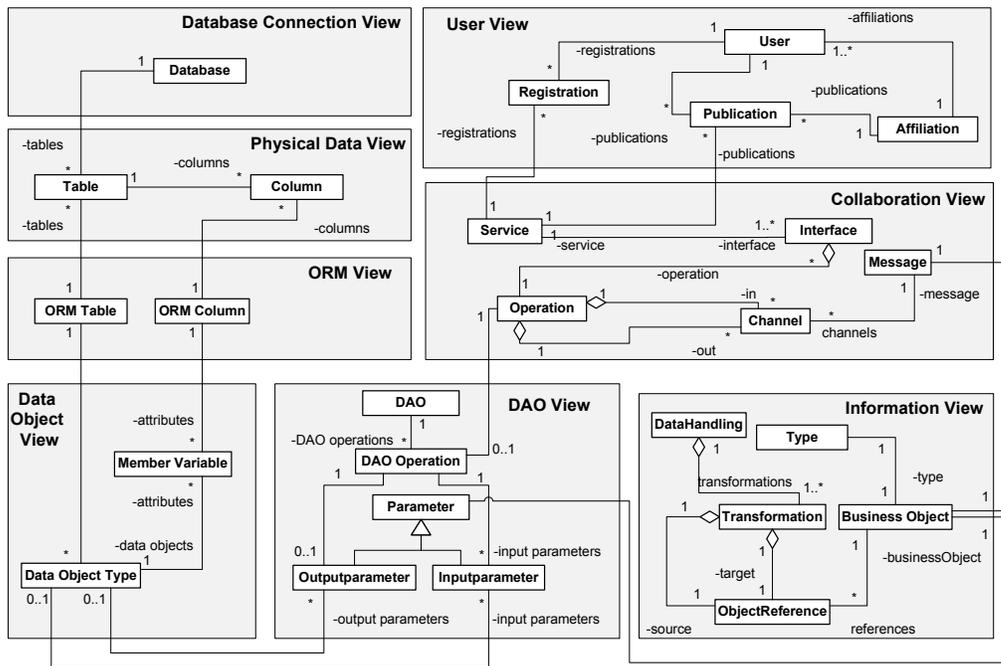


Figure 6: DAS repository View model

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 are typically based on ORM frameworks, our model provides an object-relational mapping of the `Data Object Type` to a database table (`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. [60]. 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 6, 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.

6. Tooling: The View-based Repository Client

Our view-based repository client prototype, shown in Figure 7, has been implemented as an Eclipse Plugin to comfortably supporting developers in modeling DAS. In the following we describe our view-based repository client in more detail.

6.1. Using the View-based Repository Client

The view-based repository client accesses the DAS repository by invoking its services. A detailed description of the service interface is given in Section 5.2. The UML activity diagram displayed in Figure 8 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, consists of several Eclipse views. In order to connect the Eclipse views in Figure 7 to the activities of Figure 8, the Eclipse views and the activities are labeled with corresponding numbers. In the following, 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 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

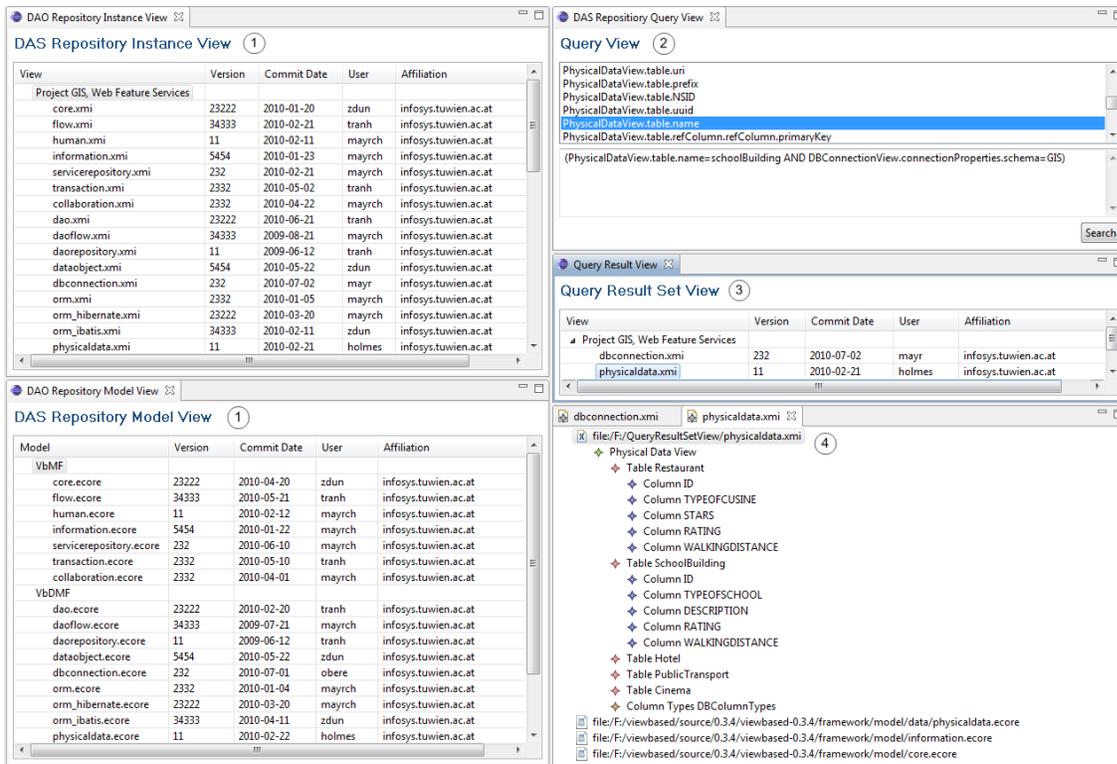


Figure 7: View-based Repository Client GUI (Eclipse Plug-in)

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 [15]. Up-to-now, our prototype view-based repository client only supports 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%).
- *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

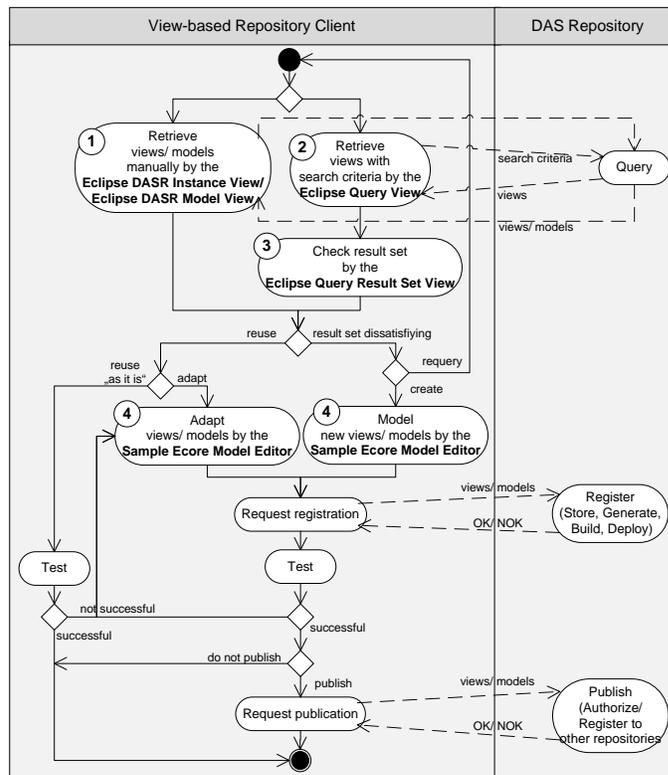


Figure 8: DAS repository: Activity Flow Diagram

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. The view-based repository client features the *Eclipse-embedded Sample Ecore Model Editor* that allows developers for comfortably specifying new view model instances. We provide an overview of our modeling framework in Section 3. 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 4). 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 oAWs Xpand language 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 View. As the DAO interfaces contain no ORM details, DAO interfaces are automatically generated simply from the DAO View and the Data Object View.

- *Test views/ models:* In order to test the views, they have to be deployed on a test environment. After successfully testing the views, the deployed DAS can be invoked by a business process application. If the test fails, developers 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 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 (see Figure 4) 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.

6.2. 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.

6.2.1. 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 [11]. 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 6.2.2, we provide some GUI support.

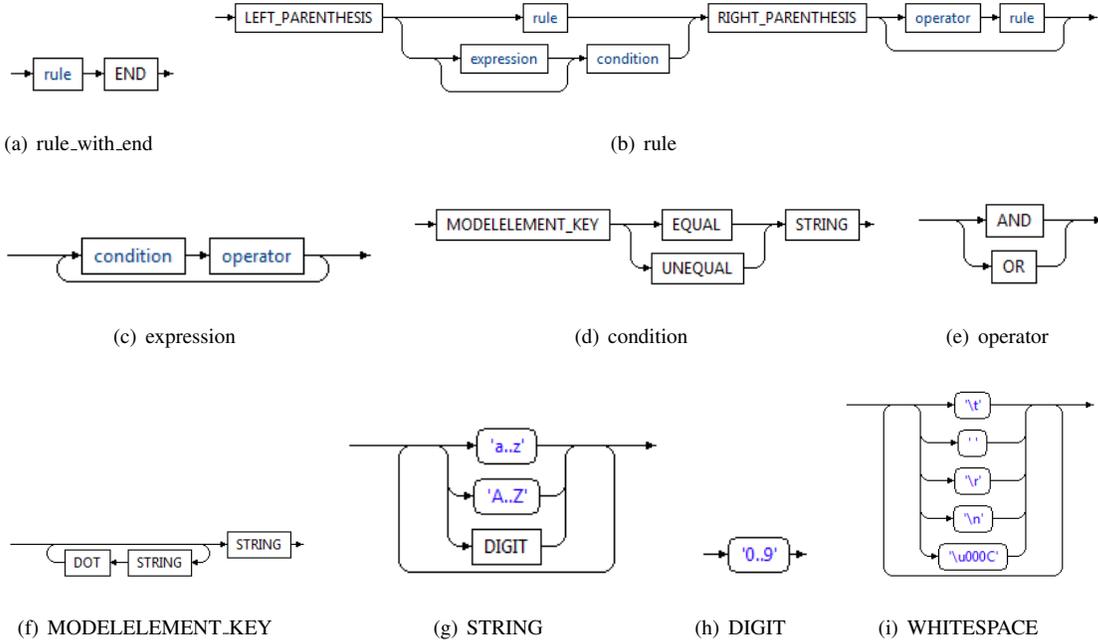


Figure 9: Query Language Definition in BNF Notation

We developed our Query Language using the Eclipse ANTLR Plug-in [23]. Figure 9 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. [23]. As shown, our language consists of simple conditions (see Figure 9(d)) and boolean operators (see Figure 9(e)). As illustrated in Figure 9(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 9(b)). Finally, the rule `rule_with_end` of Figure 9(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. A project is a set of views or models belonging together. 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.

6.2.2. 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 synonyms and of ambiguous words [26]. One way of addressing these issues is to limit the vocabulary for classifying software components, and to only allow queries drawn from this controlled vocabulary [26]. Hence, our DAS repository client proposes view model elements used for the search. The following Table 2 shows an extract of these view model elements. These view model elements are generated by the 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 2: Model Element generator: Result extraction

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

Function KeywordGenerator

```

Input: Document model
Input: Node currentNode
Input: String output
if (isEClassifier(currentNode) OR isRootNode(currentNode)) then
    HashMap < String, Node > hashMapAllNodes = getChildNodesAndInheritedChildNodes(model, currentNode);
    StringOutputPrefix = output;
    foreach String elementName ∈ hashMapAllNodes do
        Node childNode = hashMapAllNodes.get(modelName);
        if (NOT isChildNode.getNodeName().equals("#text")) then
            if (NOT isReference(childNode)) then
                if (NOT isEClassifier(childNode)) then
                    if (isRootNode(currentNode)) then
                        output = getNodeOutputString(outputPrefix, childNode);
                        KeywordGenerator(model, childNode, output);
                    else
                        output = getNodeOutputString(outputPrefix, childNode);
                        System.out.println(output);
                else
                    output = getNodeOutputString(outputPrefix, childNode);
                    Node refNode = getReferenceNode(modelName, childNode);
                    if (NOT refNode == null) then
                        KeywordGenerator(getModelByName(modelName), refNode, nodeOutputName);

```

In contrast to extracting elements from view models, extracting model elements from the underlying Ecore meta-model of the Eclipse Modeling Framework (EMF) [15] is much simpler. The reason for this is, that the elements have only be extracted from one Ecore meta-model instead of from several related view models.

7. Case Study

7.1. Business environment

In this case study we show how our approach can be applied to Geographic Information Systems (GIS) [29]. GIS are large-scale information systems making huge amount of spatial as well as non-spatial data available over the Internet [57]. A GIS data model usually consists of a large number of entities [9]. 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 [43] 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 [44]. WFS allow a client to retrieve and update spatial and non-spatial geographic data, encoded in Geography Markup Language (GML) [46], 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 [44], 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 is able to provide a specification of certain feature instances by certain spatial and non-spatial search criteria. Optionally a WFS can provide a `Transaction` operation in order to service feature modifications.

7.2. 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 paper:

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 paper, manages DAS meta-data 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 meta-data.
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 10, business process applications can find suitable WFS by querying a web catalogue service [45]. The web catalogue service manages WFS meta-data 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 is 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

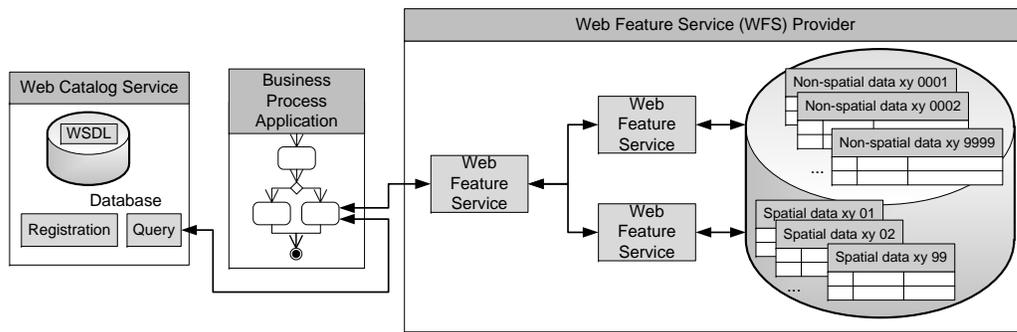


Figure 10: Case Study: WFS in OpenGIS Service Architecture

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 View, shown in Figure 11, 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 [58]. 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 (see Figure 4). 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 4) so that they can query them.

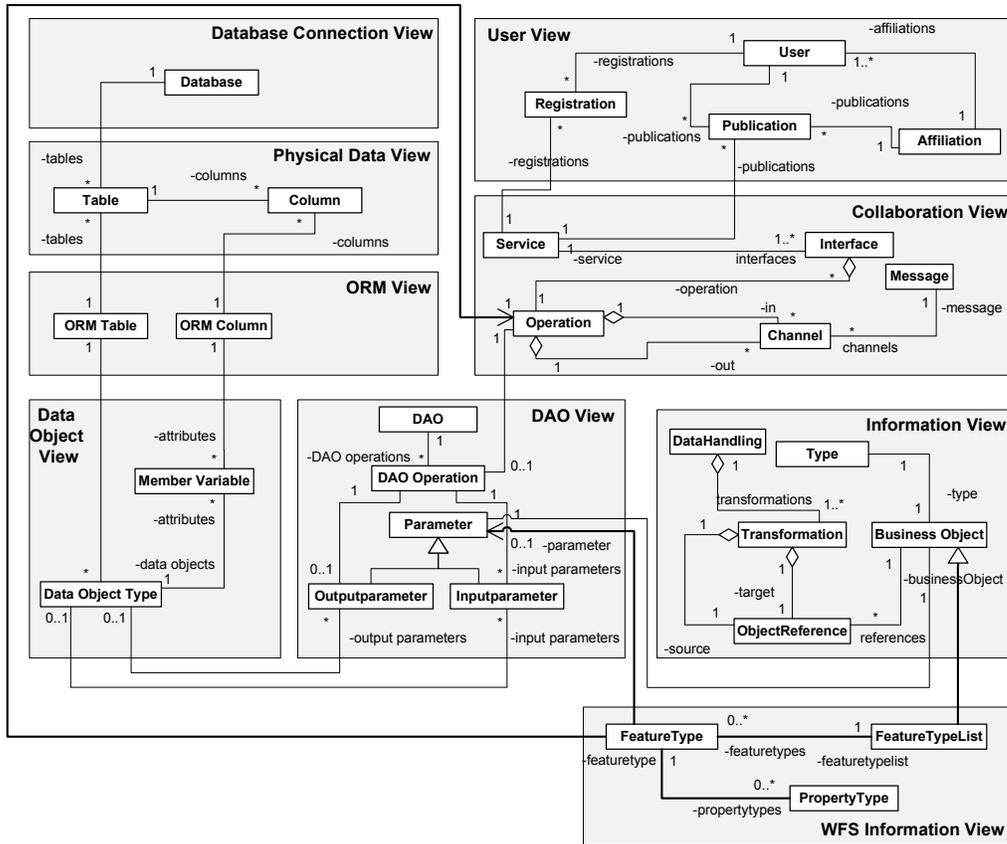


Figure 11: Case Study: Extending VbDMF by the WFS Information View

7.3. 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, 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 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 8 we illustrate further use cases in order to quantitatively evaluate our approach.

8. 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.

According to the analysis of Banker et al. [5] 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, improved product performance provides maximum satisfaction to the customers [36]. 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 [33]: an XSD model mapping approach and a domain model mapping approach. According to [33], an XSD model mapping approach should be preferred to using 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 Vb-DMF models displayed in Figure 6 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 data access in service-oriented environments in a large enterprise and secondly from analyzing WFS of commercial [18] and open-source [31] GIS. They demonstrate how the relationships between the WFS, the DAOs, and the data storage schemes can be exploited to query data from the DAS repository.

A Java-based test client application performs each use case query by invoking the DAS repository's query service by the 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 of 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 database developers need to know which WFS access a certain column of a table, 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): Database developers use this query, when they want to understand which WFS features access which database tables. In order to process this query, database 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 feature type and a table and the mapping between a feature property 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 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.

Table 3: 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

Test Requirements. Table 3 shows the numbers of data rows imported into each of the tables used for the measurement. According to Table 3, 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. A table is mapped by ORM table to exactly one data object type. A table 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 operation. 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 3 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 4: 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 4 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.

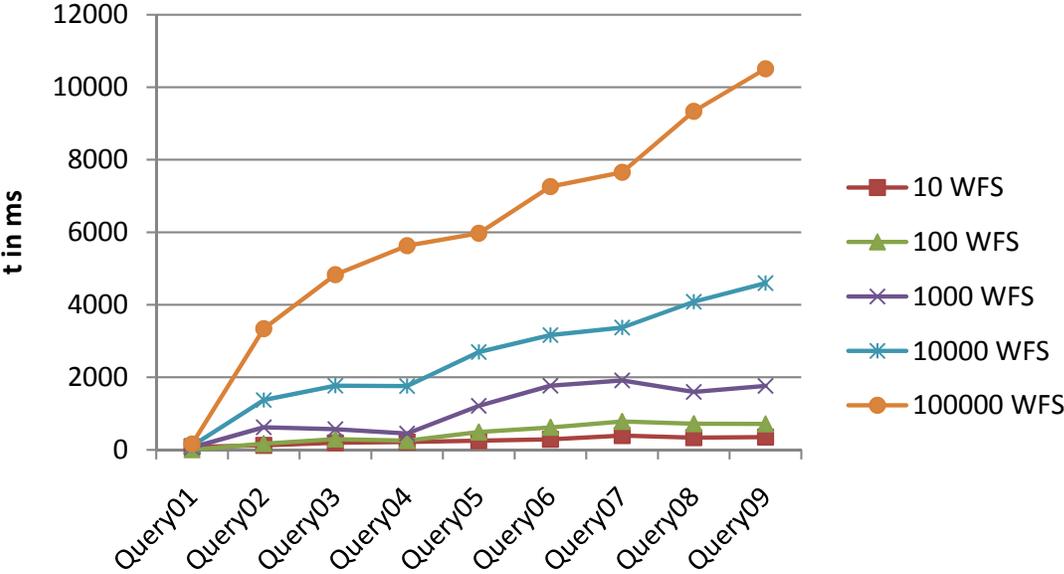


Figure 12: Experiment Result: Response time against number of table joins

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 12 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
- Randomized search criteria can lead to faster or slower results
- Different queries with different tables joining various numbers of rows as described in Table 3

As shown in Figure 12, 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. GIS usually manage several thousand features provided by several hundred WFS. As shown in Figure 12, 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 13 and Figure 14 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 12, the x-axis displays the number of WFS. As shown in Figure 14, 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.

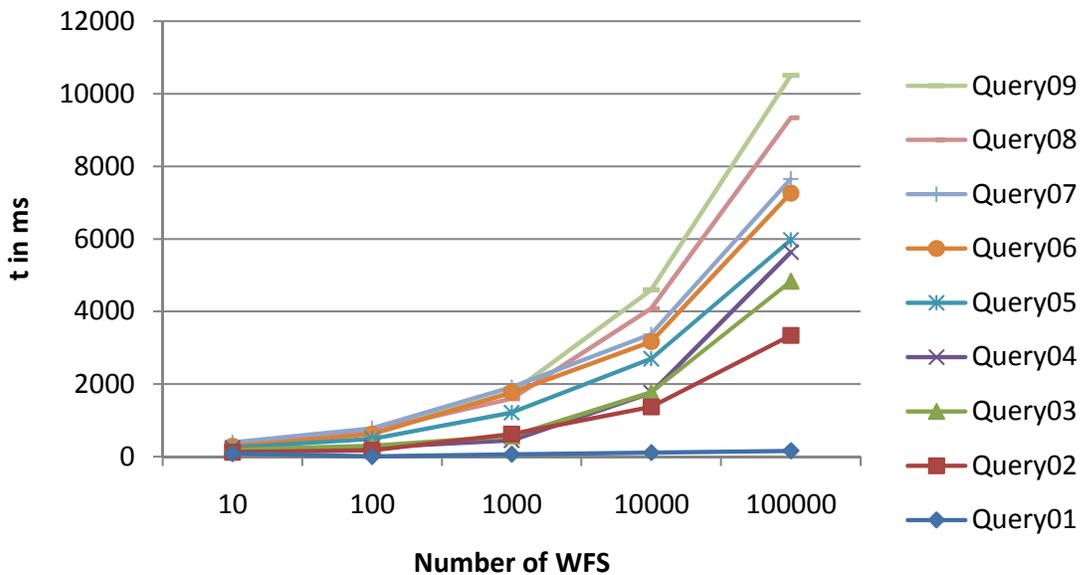


Figure 13: Experiment Result: Response time against number of WFS

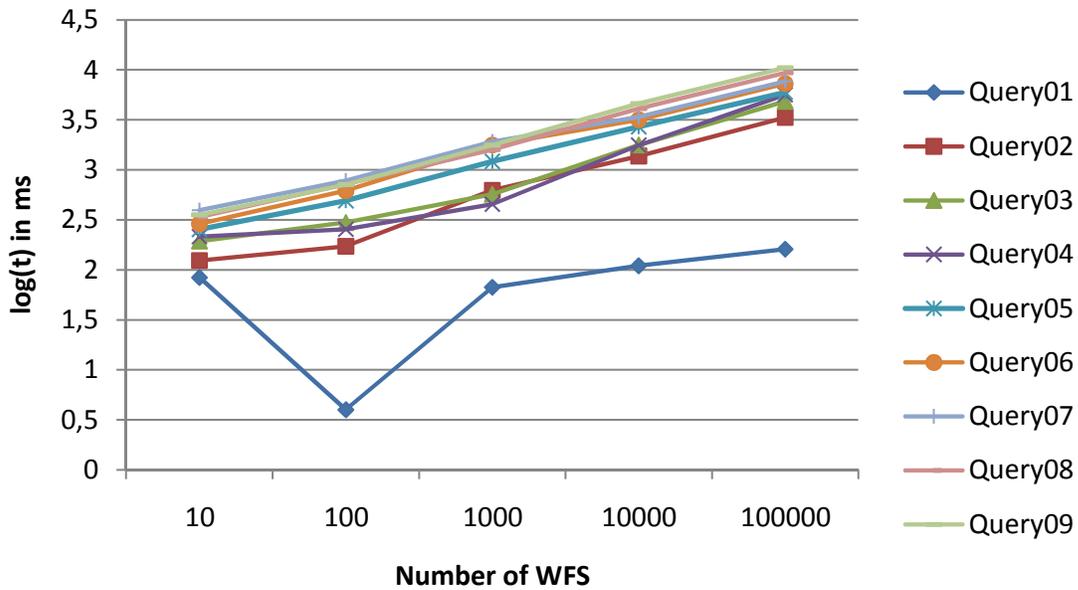


Figure 14: Experiment Result: Logarithmic response time against number of WFS

9. Conclusion and outlook

In this paper we identified a documentation gap between the data access services (DAS), the underlying DAS implementations such as data access objects (DAO), and the data storage schemes. With our approach, we mainly improve documentation of the relationships between DAS, DAOs and data storage schemes and thus contribute 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 meta-data artifacts. However, further work is necessary to coping with other important repository's requirements such as event notification, configuration control, and security. As the tools are what

gives value to a repository [6], we continue focusing on developing suitable tool chains for DAS repositories. 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. 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. Finally, runtime aspects such as dynamic invocation of DAS need to be discussed and defined.

Acknowledgement. This work was supported by the European Union FP7 project COMPAS, grant no. 215175.

References

- [1] X. L. 0005, K. Maly, M. Zubair, and M. L. Nelson. Repository synchronization in the oai framework. In *JCDL*, pages 191–198, 2003.
- [2] E. Al-Masri and Q. 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.
- [3] AndroMDA. EJB3 Cartridge. <http://web.aanet.com.au/persabi/andromda/>, Aug 2007.
- [4] K. Ballinger, P. Brittenham, A. Malhotra, W. A. Nagy, and S. Pharies. Web services inspection language (ws-inspection) 1.0. <http://www.ibm.com/developerworks/library/specification/ws-wsinspect/>, Nov 2001.
- [5] R. D. Banker, R. J. Kauffman, and D. Zweig. Repository evaluation of software reuse. *IEEE Trans. Software Eng.*, 19(4):379–389, 1993.
- [6] P. A. Bernstein and U. 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.
- [7] H. Bounif and R. Pottinger. Schema repository for database schema evolution. In *DEXA Workshops*, pages 647–651. IEEE Computer Society, 2006.
- [8] BrainML. Neurodatabase construction kit, repository server. <http://brainml.org>, Retrieved January, 2009.
- [9] H. Calkins. Entity relationship modeling of spatial data for geographic information systems. In *International Journal of Geographical Information Systems*, January 1996.
- [10] M. Cannataro, D. Talia, and P. Trunfio. Distributed Data Mining on the Grid. *Future Generation Computer Systems (FGCS)*, 18(8):1101–1112, 2002.

- [11] Y. Chen, W. W. 0011, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *SIGMOD Conference*, pages 1005–1010, 2009.
- [12] L. Clement, A. Hately, C. von Riegen, and T. Rogers. UDDI Version 3.0.2, UDDI Spec Technical Committee Draft. http://www.uddi.org/pubs/uddi_v3.htm, Oct 2004.
- [13] J. 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.
- [14] Z. Du, J. Huai, and Y. Liu. Ad-uddi: An active and distributed service registry. In *TES*, pages 58–71, 2005.
- [15] Eclipse. Eclipse modeling framework (EMF). <http://www.eclipse.org/emf/>, 2006.
- [16] Eclipse. Eclipse CDO. <http://wiki.eclipse.org/CDO>, CCopyright 2009.
- [17] Eclipse. Eclipse Modeling Framework Project. <http://www.eclipse.org/modeling/emf/>, Retrieved December, 2008.
- [18] ESRI. esri arcgis. <http://www.esri.com/software/arcgis/index.html>, February 2011.
- [19] R. Fang, L. Lam, L. Fong, D. Frank, C. Vignola, Y. Chen, and N. Du. A version-aware approach for web service directory. In *ICWS*, pages 406–413, 2007.
- [20] Fornax-Platform. Cartridges. <http://www.fornax-platform.org/cp/display/fornax/Cartridges>, Aug 2006.
- [21] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [22] Hibernate. Hibernate. <http://www.hibernate.org>, 2006.
- [23] <http://antlr3ide.sourceforge.net/>. Antlr ide. an eclipse plugin for antlrv3 grammars. An eclipse plugin for ANTLRv3 grammars, July 2010.
- [24] Ibatis. Ibatis. <http://www.ibatis.org>, 2006-2007.
- [25] IBM. Websphere service registry and repository. <http://www-01.ibm.com/software/integration/wsrr/>, March 2011.
- [26] T. Isakowitz and R. J. Kauffman. Supporting search for reusable software objects. *IEEE Transactions on Software Engineering*, 22:407–423, 1996.
- [27] M. B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. Packt Publishing, 2006.

- [28] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Softw.*, 20(6):35–39, 2003.
- [29] P. A. Longley, M. Goodchild, D. J. Maguire, and D. W. Rhind. *Geographic Information Systems and Science*. John Wiley & Sons; 3rd Revised edition edition, August 2006.
- [30] S. A. Ludwig and S. M. S. Reyhani. Semantic approach to service discovery in a grid environment. *Web Semant.*, 4(1):1–13, 2006.
- [31] D. Masclet. Gisgraphy. <http://www.gisgraphy.com/index.htm>, August 2010.
- [32] C. Mayr, U. Zdun, and S. Dustdar. Model-driven integration and management of data access objects in process-driven soas. In *ServiceWave '08: Proceedings of the 1st European Conference on Towards a Service-Based Internet*, pages 62–73, Berlin, Heidelberg, 2008. Springer-Verlag.
- [33] C. Mayr, U. Zdun, and S. Dustdar. Reusable architectural decision model for model and metadata repositories. In *FMCO*, pages 1–20, 2008.
- [34] N. Milanovic, R. Kutsche, T. Baum, M. Carlsburg, H. Elmasgünes, M. Pohl, and J. Widiker. Model&metamodel, metadata and document repository for software and data integration. In *MoD-ELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 416–430, Berlin, Heidelberg, 2008. Springer-Verlag.
- [35] J.-K. Min, C.-H. Lee, and C.-W. Chung. Xtron: An xml data management system using relational databases. *Information & Software Technology*, 50(5):462–479, 2008.
- [36] K. B. Misra. *Handbook of Performability Engineering – Quality Engineering and Management*. Springer London, 2008.
- [37] NetBeans Community. Metadata repository (mdr). <http://mdr.netbeans.org/>, Retrieved January, 2009.
- [38] S. D. Network. Core J2EE Pattern Catalog. <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>, Copyright 1994-2008 Sun Microsystems, Inc.
- [39] H. W. Nissen and M. Jarke. Repository support for multi-perspective requirements engineering. *Inf. Syst.*, 24(2):131–158, 1999.
- [40] B. Nuseibeh, A. Finkelstein, and J. Kramer. Method engineering for multi-perspective software development. *Information & Software Technology*, 38(4):267–274, 1996.
- [41] 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.

- [42] OASIS/ebXML Registry Technical Committee. Registry Services Specification v2.0. <http://www.ebxml.org/specs/ebrs2.pdf>, Dec 2001.
- [43] Open Geospatial Consortium, Inc. OpenGIS Service Architecture. <http://www.opengeospatial.org/standards/as>, January 2002.
- [44] Open Geospatial Consortium, Inc. OpenGIS Web Feature Service (WFS) Implementation Specification. <http://www.opengeospatial.org/standards/wfs>, May 2005.
- [45] Open Geospatial Consortium, Inc. OpenGIS Catalogue Services Specification. <http://www.opengeospatial.org/standards/specifications/catalog>, February 2007.
- [46] Open Geospatial Consortium, Inc. OpenGIS Geography Markup Language (GML) Encoding Standard. <http://www.opengeospatial.org/standards/gml>, August 2007.
- [47] openArchitectureWare. oAW. <http://www.openarchitectureware.org>, Aug 2002.
- [48] E. Ort and B. Mehta. Java Architecture for XML Binding (JAXB). <http://java.sun.com/developer/technicalArticles/WebServices/jaxb/>, March 2003.
- [49] L. 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.
- [50] M. P. Robillard and F. 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.
- [51] C. Sant’anna, A. Garcia, C. Chavez, C. Lucena, and A. v. von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.
- [52] T. Schwede, J. Kopp, N. Guex, and M. C. Peitsch. Swiss-model: An automated protein homology-modeling server. *Nucleic Acids Res*, 31(13):3381–3385, July 2003.
- [53] A. Shaikh Ali, S. Majithia, O. F. Rana, and D. W. Walker. Reputation-based semantic service discovery: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(8):817–826, 2006.
- [54] L. Steller, S. Krishnaswamy, and J. Newmarch. Discovering relevant services in pervasive environments using semantics and context. In *IWUC*, pages 3–12, 2006.
- [55] H. Tran, U. Zdun, and S. Dustdar. View-based and model-driven approach for reducing the development complexity in process-driven SOA. In W. Abramowicz and L. 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.

- [56] M. Turner, D. Budgen, and P. Brereton. Turning software into a service. *Computer*, 36:38–44, 2003.
- [57] L. Vaccari, P. Shvaiko, and M. Marchese. A geo-service semantic integration in spatial data infrastructures. *International Journal of Spatial Data Infrastructures Research*, 4:2451, 2009.
- [58] V. M. P. Vidal, F. C. Lemos, and F. Feitosa. Translating wfs query to sql/xml query. In *GeoInfo*, pages 174–190, 2005.
- [59] M. Völter and T. Stahl. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.
- [60] W3C. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wSDL>, January 2010.
- [61] F. Zhu, M. Turner, I. Kotsiopoulos, K. Bennett, M. Russell, D. Budgen, P. Brereton, J. Keane, P. Layzell, M. Rigby, and J. 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.