# Reusable Architectural Decision Model for Model and Metadata Repositories

Christine Mayr, Uwe Zdun, and Schahram Dustdar

Distributed Systems Group
Information System Institute
Vienna University of Technology, Austria
christine.mayr@inode.at, {zdun,dustdar}@infosys.tuwien.ac.at

**Abstract.** Models are gaining importance in software development, for instance in the MDD field, as well as in other disciplines such as biology and physics. Hence, tool support is needed to manage these models and metadata about the models. Model repositories support this trend by managing these model artifacts. While setting up model and metadata repositories, architects have to make several fundamental design decisions and balance various forces. In this paper we describe reusable knowledge in form of reusable architectural decisions for IT-architects in setting-up, planning, and developing model and metadata repositories, as well as the main decision drivers. Our decisions are documented in a reusable architectural decision model that can be instantiated for a concrete system. It also supports a lightweight approach to architecture documentation. A case study illustrates the decisions made when setting up our own data access object model repository by walking through the reusable architectural decision model.

## 1 Introduction

Today many systems are modeled with precisely specified and detailed models. Reasons are among others the increasing support for model interoperability between modeling tools [1] and the increasing use of model-driven development (MDD) [2, 3]. In MDD many tools in a tool chain must work on a set of models, and they must be able to import models developed with external modeling tools.

Model repositories [4, 5, 6, 7] support this trend by managing modeling artifacts, such as models, model instances, model relationships, and so on. A model repository enables modelers to create, retrieve, update, and delete modeling artifacts, and to query for them. Usually additional metadata about the modeling artifacts can be stored and used in the queries. Some repositories are even pure metadata repositories. In addition, model repositories can support extra functionality, such as versioning support, security functions, or storing of related source code artifacts.

Model repositories should often be optimized for the kind of modeling artifacts they store and the task they should fulfill. For instance, usually custom, model-aware queries should be provided that are simplified or more powerful compared to standard queries, such as SQL queries, because they can make use of the information in the modeling artifacts. Model repositories are often realized on top of existing basic technology such

F.S. de Boer et al. (Eds.): FMCO 2008, LNCS 5751, pp. 1–20, 2009.

as databases, but it is not enough to simply store the models in and retrieve them from such a basic technology. In this context, a number of recurring design decisions must be made. In this paper, we propose a reusable architectural decision model that describes these design decisions in a reusable fashion, so that they can be applied step-by-step for new model repository projects. Our research results are based on field notes and observations from our own model repository projects, a detailed analysis of existing model repository projects (both open source and commercial), and interviews and discussions with other model repository developers.

In this paper we provide architectural decision-support for architects in finding a suitable solution to resolve fundamental design problems arising when planning and setting-up model and metadata repositories. For each decision we present recommendations which alternative to choose depending on certain requirements and boundary conditions. Some of our decisions might be intuitively decided in a suitable way by architects. However, other decisions might be skipped or decided in a non-optimal way because of missing knowledge of alternatives and consequences. Our approach mainly aims at decreasing the costs and impact of making wrong decisions related to setting-up model and metadata repositories. In addition, our approach can be used as a lightweight approach to architecture documentation: If the reusable architectural decision model is used to make decisions, only a reference to the decision model is needed to document an architectural decision instead of documenting the whole decision as well as the rationale.

Our paper is structured as follows: First, we define the terms repository, metadata repository, and model repository in Section 2. In Section 3 we introduce reusable architectural decision models as the background of our work. Section 4 provides detailed specifications of the architectural decisions and describes the dependencies between them. We illustrate the applicability of our approach through a case study in the area of modeling jurisdictional provisions in the context of a district court, described in Section 5. Section 6 discusses related work, and finally Section 7 concludes this paper.

## 2 Repository, Metadata Repository, and Model Repository

Before we go deeper into modeling architectural decisions of a model repository or metadata repository, we would like to define the terms *repository*, *metadata repository*, *model repository* and *model and metadata repository*, as these are forming the basis for our work. The field of repositories is currently a popular area of research. Therefore the following definitions are not exhaustive with regard to a full functional and non-functional requirements specification of a repository. These nominal provisions rather point out those characteristics of a repository we in particular focus on in this paper.

We define a *repository* as a central accessible component storing information about reusable artifacts [8]. Examples of these artifacts are source code, documents, and special-purpose models such as data models for defining the relationships between objects in object-oriented environments, models for MDD [2], biology models [9], and so on. Furthermore, a repository has to provide the means to query these information artifacts and metadata about these information artifacts respectively according to certain search criteria. In many cases, querying is performed using some query language.

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

When a repository provides models and/or model instances such that it either stores models and/or model instances as its artifacts or provides these models and/or model instances stored at other locations, we refer to a repository as a *model repository*. Usually, a model repository additionally provides metadata of models or model instances. Hence, we refer to a repository that provides meta-data of models and/or model instances as a *model and metadata repository*.

## 3 Reusable Architectural Decision Models

According to Taylor and van der Hoek [10], as well as Jansen and Bosch [11], software architecture is a set of principal design decisions governing a system. During a software system's design phase, architects have to make numerous decisions for organizational and business issues, for matters of broad and detailed design, and for technologies [12]. We refer to a design decision using the term *architectural decision*, if firstly it affects either the architecture of a system or the role of the architect. Secondly, the architects of the system see those decisions as *principal* decisions. The main argument for using architectural decision modeling is that such principal decisions should not get lost.

*Architectural decision models* are used to document architectural decisions [11, 13, 14]. These architectural models capture selected decision options and justifications for these decisions. In industry, architects often do not attach great value to decision modeling, and, if it is performed at all, architectural decision modeling is usually done in retrospect. Thus, architectural decision models cannot solve all problems of lacking documentation [15, 14]. Many techniques such as text templates and tool support have been proposed, but until now they have not become broadly adopted in practice [12].

Reusable architectural decision models proposed by Zimmermann et al. [16, 12] focus on solving these problems. A *reusable architectural decision model* enhances the basic decision model by steering the architectural decision making activities [12]. Reusable decision models are closely related to software pattern concepts (see [15]). For instance, Zimmermann et al.'s approach uses the resuable decision models for pattern selection. The advantage of this approach is that a decision model that is based on patterns does not have to copy the pattern text and hence is easier to create than a self-contained decision model.

In this paper, we describe a reusable architectural decision model for model and metadata repositories. Each *architectural decision* is characterized by a *decision name*. In our model, the decisions either have a number of *alternatives* or *options* for which the architect can decide. Some alternatives or options have *variants*, which can be selected, too. For each decision, we describe the *forces* or *decision drivers* that must be considered when selecting an alternative or option. Usually, the different alternatives and options have different *consequences* with regard to the forces. To illustrate the alternatives or options, we describe a few *known uses*. Finally, decisions have *relationships* to

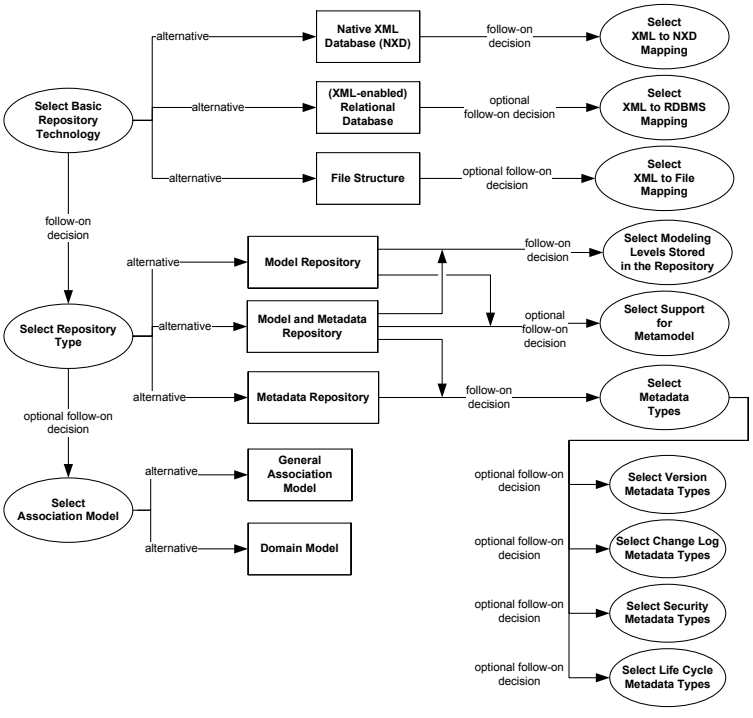



**Fig. 1.** Dependencies between architectural decisions

other decisions. For instance, a decision can be a follow-on decision to another decision, if a specific alternative or option is chosen.

## 4 Architectural Decisions

In this section, we describe architectural decisions architects must make for planning, setting-up, developing, and installing a model repository. In particular, we focus on the underlying data model design – the core of a model repository. At first, we give a short overview over these decisions and the dependencies between them (see Figure 1). Subsequently, we present each of these decisions in detail.

The decision model is distilled from our experiences, our study of other projects (both open source and commercial), as well as the documented experiences of others. Please note that the decisions and their alternatives are for this reason not exhaustive.

- *Select Basic Repository Technology*: Usually, one of the first decisions is which basic technology should be used for the repository. Depending on the types and amounts of models or metadata to be stored, either an XML database, a specific file structure, or a standard relational database are alternatives.
- *Select XML to NXD Mapping*: When architects decide for an XML database, they can select between two basic mapping alternatives, namely an XSD model-based and a text-based approach.

- *Select XML to RDBMS Mapping*: When architects choose an RDMBS, an important follow-on decision is how to map the XML documents to the database, namely by a domain model mapping or an XSD model mapping.
- *Select XML to File Mapping*: When architects decide for a file storage solution, they can select between three basic mapping alternatives, namely a XSD model-based, a domain model-based and a simple text-based approach.
- *Select Repository Type*: Depending on important decision drivers such as searching capabilities and data categorization, architects can decide for a Model Repository, a Metadata Repository, or a Model and Metadata Repository.
- *Select Support for Metamodel*: When architects decide for storing models by selecting the Model Repository and Model and Metadata Repository respectively, they optionally can choose a Metamodel that specifies the elements of the stored models.
- *Select Modeling Levels Stored in the Repository*: Architects have to select the modeling levels such as models, model instances, source code, and runnable code to be stored in a Model Repository or a Model and Metadata Repository.
- *Select Metadata Types*: In case a Metadata Repository or a Model and Metadata Repository is used, architects can select model-independent metadata such as version information, ownership, affiliations, and security data.
- *Select Version Metadata Types*: An optional follow-on decision of selecting necessary metadata types is choosing an adequate version granularity. Versioning can be either settled on the model or model element level.
- *Select Change Log Metadata Types*: According to the decision of selecting version metadata, architects have to decide whether to add change log metadata to either the model or model element level.
- *Select Security Metadata Types*: Architects can choose among several security metadata options. Unlike the decisions described before, security metadata does not solely focus on several artifacts, but on mechanisms to secure the whole repository.
- *Select Life Cyle Metadata Types*: Architects can opt for a life cycle manager, that can determine if a requested action is allowed dependent on the current state.
- *Select Association Model*: This decision deals with whether to model relationships in the domain models themselves or using a general association model [17].

### 4.1 Architectural Decision: Select Basic Repository Technology

A fundamental task of a model repository architect is to choose a basic storage technology for the repository. As illustrated in Figure 2, there are three basic alternatives for storing artifacts: Native XML Databases (NXD), (XML-enabled) RDMBS, and a File System using a specific file structure. RDF triple stores are a popular variant of NXD.

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

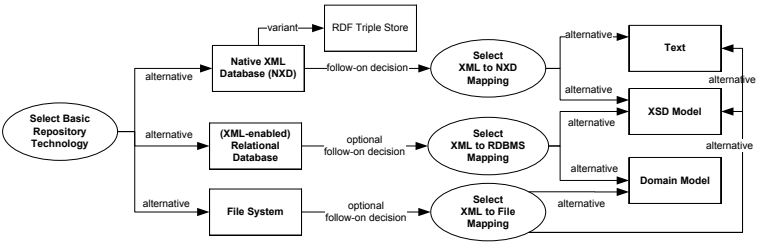



**Fig. 2.** Architectural Decision: Select basic Repository Technology

to use a repository based on *standard technology*, as standard interfaces often ease the integration. For standard technologies, often a number of tools and IDE plugins exist, which help developers and partners to work with the repository.

There are a number of follow-on decisions related to mapping XML to one of these storage alternatives. Although we mention alternative exchange formats such as objects of a programming language (e.g., as possible in EMF [18]), because XML is the common model data exchange format, in the following we focus only on an in-detail description on XML model exchange format mappings.

Each of these approaches has its own advantages and limitations [19]. Particularly with regard to throughput and huge amount of data, a NXF system may work best, because no mapping process from XML files to database schemes is required [19]. Furthermore most native XML databases support sophisticated full-text searches. However, due to the document-centric approach, complex queries can have longer response times compared to (XML-enabled) RDBMS systems [20]. One known use is XTC, the XML Transformation Coordinator for XML Document Transformation Technologies [21].

Relational databases provide both maturity, scalability, portability, and stability [19, 22], and they are the RDBMS that are probably most widely used today [19]. Known uses of model repositories based on RDMBS are the SWISS–MODEL Repository [23] for three-dimensional comparative protein structure models and the BrainML Model Repository [24] storing neuroscience data.

Alternatively, especially for small amounts of data, architects could choose a simple file structure as repository storage. For this, one of many known uses is the CellML Model Repository [9] for storing and exchanging computer-based mathematical models. Of course, when using a file storage, searching a large amount of data, could be rather inefficient in comparison to using either a NXD system or RDBMS. However, for repositories with only small amount of data, this might be the simplest and most appropriate solution. In particular when using proprietary file formats, the repository can be set-up quickly, because no data mapping is required.

### 4.2 Architectural Decision: Select XML to NXD Mapping

Provided that architects opt to use a Native XML database, they can decide between two basic storing alternatives (see Figure 2). Either the entire XML document can be stored in *Text* format or the XML document can be modeled as DOM and mapped to *XSD*

*Model* objects such as Elements, Values, etc. [19]. In the former case the database or file managing component has to manage indexes to improve performance on its own. In the latter case, XML documents can be stored as type-annotated trees on disk pages [25]. These database trees are indexed with path-specific indexes, and can be queried with XQuery and SQL/XML [25].

Whether to use a text-based or an XSD model-based mapping depends on the required *performance* and on the *effort* to establish the system. There are many NXD systems both commercial and open-source. Most XML databases such as DB2 [25] support the *XSD Model* for Mapping XML to corresponding tree structures in NXD [20].

### 4.3 Architectural Decision: Select XML to RDBMS Mapping

Provided an RDMBS database is selected as the basic repository technology and the raw models are provided in XML format, an important follow-on decision is how to resolve the conflict between the hierarchical nature of an XML data model and the row and column nature of a relational data model [19, 20]. Architects can mainly choose between two alternatives: They can either decide to map the *Domain Model* elements to a database schema or use an *XSD model* approach by mapping standard *XML model* elements to RDBMS. By using the *Domain Model* mapping approach, a separate table is generated for each domain model element. In contrast, the *XSD Model* mapping approach is characterized by a lesser number of resulting tables and columns, because unlike the *Domain Model* approach, several XML elements are combined into a single table. Moreover, the resulting RDBMS schema, here, can either be generated from an XSD or from a DTD. Algorithms for mapping XML data to relational data can be found in [20]. See [26] for a comparison of the most cited and DTD-independent methods in terms of resource usage and query response times.

Many commercial XML-enabled database systems such as SQL Server and Oracle support both the *XSD Model* and the *Domain Model* mapping. In the latter approach the existing data model is extended to an additional XML data type [20].

Decision drivers are both *performance* and the *effort* to accomplish the mapping. In case neither an XSD nor a DTD exists, architects should decide to use the *XSD Model* mapping approach. Additionally, this approach can reduce the number of join operations incurred during query operations [19]. Florescu's and Kossmann's work [27] shows that even the simplest and most obvious approaches provide a good performance. Thus, in most cases, we would clearly recommend to use the *XSD model* mapping approach, especially if performance is the most important decision driver.

### 4.4 Architectural Decision: Select XML to File Mapping

In case architects decide to use an appropriate file system structure and the models are stored as XML documents, they can select among three basic storing alternatives (see Figure 2). The file itself can contain the entire XML document as *Text*, the XML document can be separated according to the *XSD* model, or the document can be split into several files according to its *Domain Model*.

The advantages and disadvantages for using the XSD model-based or the Domain model-based approach were already discussed in Section 4.2 and Section 4.3

respectively. The obvious advantages of the text-based alternative are simplicity and the low effort to establish the system. Thus which alternative to use depends on the required *performance* and on the *effort* to establish the file system storage.

### 4.5 Architectural Decision: Select Repository Type

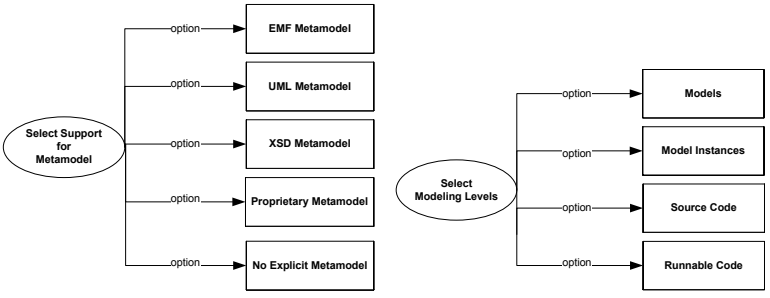
Depending on the repository's functional requirements, models, model instances, and/or metadata must be stored in a model repository. As already defined in Section 2, we can distinguish three alternative repository types: *Model Repository*, *Metadata Repository*, and *Model and Metadata Repository*. Figure 1 depicts these alternatives. Most repositories use metadata to describe general characteristics such as version information, user information, and security data. In contrast to metadata, models contain domain-specific elements. Some metadata, such as version information, is linked to specific models as add-on data, other metadata, such as user authorization data, can be considered as general repository data that is not linked to specific model data. In Section 4.8 we focus on selecting adequate metadata types.

The decision drivers for storing models in the repository are mainly *functional requirements*. Examples are: An important decision for architects is if the MDD paradigm [2] should be supported using the repository architecture. When using MDD, the source code is generated from the underlying models and these models must be accessible from the repository. In Section 2 we stated that a repository should provide *query mechanisms* to search for repository artifacts according to certain search criteria. These query mechanisms are based on categorized data such as domain specific model data and repository metadata.If architects want to store non-model artifacts, in order to provide appropriate searching mechanisms, they should at least provide these artifacts with some add-on metadata. Accordingly, in case solely non-model artifacts are stored in the repository and provided with add-on metadata, architects decide in favor of a *Metadata Repository*.

Architects choose a *Model Repository* if they intend to store models in the repository and do not require additional metadata, because the domain models possibly contain part of this information. Moreover, adding special-purpose metadata such as ownership and affiliation information to repositories in small companies may not be necessary.

If more sophisticated queries about the repository artifacts are required, architects should consider storing categorized model data and thus select the *Model and Metadata Repository* alternative. A known use of a *Model and Metadata Repository* is the Data Access Object (DAO) Repository that we developed during our studies. In our case study (see Section 5) we give more details about the DAO repository by applying it to our reusable architecture decision model.

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




(a) Select Support for Metamodels (b) Select Modeling Levels Stored in the Repository

**Fig. 3.** Architectural Decisions

### 4.6 Architectural Decision: Select Support for Metamodel

Provided that architects decide for a *Model Repository* or a *Model and Metadata Repository*, they can select a metamodel for the domain models to be stored. A metamodel describes models and thus is the basis for model validation by tools. Eessaar illustrates the advantages of using metamodels [28]: Metamodels are a clear and useful supplement to textual specifications. Compared to a purely textual specifications, metamodels enable a much more compact and clear overview of the model. In addition, metamodels such as UML and EMF [18] can support visualizing models and thus ease model readability and understandability. It has also been demonstrated that a metamodel could be used to compare heterogeneous models. In the literature there are various approaches addressing the problem of integrating heterogeneous models [29, 30].

Decision drivers are both the *functional* and *technical requirements*. Firstly, architects might use an explicit metamodel if they wish to benefit from one or more of the properties described above. Secondly, technology reasons such as using MDD [2] can be a determining factor for using a metamodel. In case of MDD [2], architects profit from tool support. For instance, they can use a metamodel-based generator, such as openArchitectureWare [31], to generate source code from models specified by a corresponding metamodel such as EMF [18]. If architects do not want to profit from these functional and technical features, they can make use of a simple, but much less flexible approach: To support no explicit meta-model. That means, to hard-code the metamodel information and thus specifying a model without an underlying metamodel.

In addition to that option, in Figure 3(a) we illustrate several metamodel options among which architects can select: EMF [18], UML, XSD, and a proprietary domain meta model (see Figure 3(a)). They should choose a proprietary domain metamodel, if standard metamodels such as UML and EMF [18] do not fulfill the requirements.

A known use of using EMF [18] metamodels, is our VbMF [32] repository that we developed during our studies. A known use for a model repository that loads UML2 models into EMF is the AndroMDA's EMF UML2 Repository [33]. In contrast, the BrainML Model Repository [24] consists of a standard XML Schema, defining XML elements, and referencing other schema definitions using standard mechanisms.

### 4.7 Architectural Decision: Select Modeling Levels Stored in the Repository

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

Figure 3(b) depicts this architectural decision and its four modeling layer options. In the following we specify important decision drivers for each of these layers.

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

The next decision is whether architects should store model instances in the repository. This decision is closely related to the required *search capabilities*. Besides the desired search capabilities, another decision driver is whether to support MDD or not. In case MDD is supported, model instances rather than source code are stored by the repository because the generator can use transformations to generate the source code. In some cases, this means that the transformations for the generator should also be placed in the repository . However, even for non-model-driven projects, we recommend storing model instances, if at least simple queries to find certain source code are required.

There is also the option to store the model instances but not the models. An example of a known use that stores model instances, but no models is the Eclipse CDO Project [6]. In contrast, another known use, the Netbeans Metadata Repository (MDR) [7], stores both models and model instances.

Whether the repository should provide source code, depends both on the *technical requirements*, such as using MDD [2], and on the *development environment* and *platform* of repository users. When MDD is used, commonly *technology- and platform-independent* model instances are stored in the repository. Accordingly, on the client side, repository users can generate source code from these model instances according to their specific platform- and technology requirements. Thus, if more than one technology or platform should be supported, source code should not be stored in the repository, but generated by the repository users. Otherwise, if no technology- and platform-dependent source code generators are required, architects can decide to store the source code in the repository. In this case, generated source code can also be stored in the repository, e.g., to archive it. Alternatively, the source code can be stored in an external repository specified by references in the models (if the model instances should be aware of the source code artifacts) or appropriate metadata information (for more information about selecting metadata types please refer to Section 4.8).

The next decision architects should make is whether the repository should supply runnable byte code and how. In the following we present three alternatives: The first alternative proposes to build the source code on the client side. This alternative primarily

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

### 4.8 Architectural Decision: Select Metadata Types

Common repositories include metadata to provide additional, model-independent information of repository artifacts. Figure 4 shows a few options: Metadata can include versioning information; change log data; ownership and/or affiliation information; security data such as information on role-based access control and identity management; location information; life cycle data and data for internationalization features (see Section 4.8). In the following we give a detailed overview of each of these metadata options commonly used in repositories. Architects can use this checklist to decide whether to apply a certain metadata type or not. We have developed this checklist by studying common repositories to the best of our knowledge. However, due to the diversity of possible metadata types, the list is not exhaustive. After illustrating the checklist, in the proceeding sections (4.9, 4.10, 4.11, 4.12) we particularly focus on the follow-on decisions as well as resulting options and alternatives depicted in Figure 4.

*Version Information Metadata.* Architects have to decide whether to add version metadata or not. In the simple case, architects can opt for using no versioning. For this purpose, they solely need to provide the *most recent version* of repository artifacts.

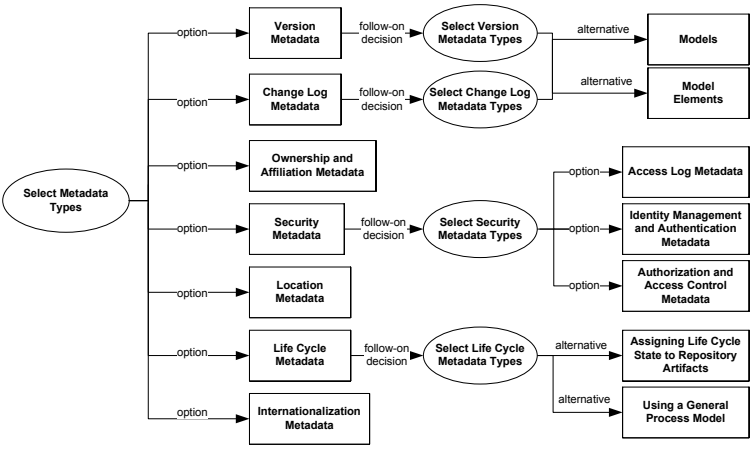



**Fig. 4.** Architectural Decisions: Select Metadata Types

Otherwise, if the repository shall support version management, the have to make the follow-on decision illustrated in Section 4.9.

*Change Log Metadata* Change log metadata can include information about which user inserted or updated a certain repository artifact. The decision whether to add change log data is based on the previous decision of adding *version information metadata*. Thus, architects can not opt for providing change log metadata, not until they decide in favor of using version metadata. In Section 4.10, we present the follow-on decision of selecting different change log metadata types.

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

*Security Metadata.* According to their security requirements, architects can choose one or more types of security metadata (see ebXML Registry Services and Protocols [17]). Please note that unlike other types of metadata, security metadata does not solely focus on several artifacts, but on mechanisms to secure the whole repository. In Section 4.11 we present some basic security options architects can install.

*Location Metadata.* Another type of metadata architects can choose is location metadata. As already mentioned in Section 4.7, source code and runnable code can be linked to models and model instances stored in other repositories. The decision drivers for deciding whether source code and runnable (byte) code should be stored in the repository itself or in an external repository are the same as those describd in Section 4.7. Besides source code and runnable code, location metadata can be important, e.g., for linking model instances or source code to specific documentation on document servers. In order to *save storage cost* and *maintainance efforts*, we recommend to decide in favor of referring to existing documentation instead of storing this information redundantly.

*Life Cycle Metadata.* A repository incorporating life cycle metadata manages all life cycle actions such as inserting, updating, deleting, and deprecating repository artifacts. Besides these basic actions, the life cycle manager can oversee further actions such as validating model instances and finally publishing changes to repository users. Depending on the current life cycle state, the life cycle manager determines if the requested action is allowed and consequently performs or rejects the action. In Section 4.12 we present the follow-on decision of selecting a suitable life cycle metadata type.

*Internationalization Metadata.* Internationalization metadata can be used for storing location-specific settings, such as different languages and coding sets. In the EBXML standard [17] internationalization metadata is defined as attributes that are I18N capable and may be localized into multiple native languages. Architects may choose internationalization metadata, if e.g. *international project members* shall access the repository or *different coding sets* shall be supported.

### 4.9 Architectural Decision: Select Version Metadata Types

When storing models, architects can decide to either add version information metadata to the whole model or to each model element. The CellML Model Repository [9] is a known use of a repository, that stores version information at the model level. If a CellML model is modified, the new updated version(s) are added to the repository and they are automatically allocated a new version number [9]. The BrainML Model Repository [24] is another known use that adds version information metadata at the model level. Version numbers start at 1 and are incremented whenever an augmented or modified version of the model is submitted. Earlier versions remain available in the repository and can be referenced by their version number to support data using them.

Standards such as UDDI [34], EbXML [17], and the Content Repository API for Java Technology of Java Specification Request (JSR) 170 [35] support adding version information to model elements. JCR consists of one or more workspaces that each consist of a tree of items representing either nodes or properties. A content repository [35] workspace that supports versioning may contain both versionable and nonversionable nodes. A known use open-source implementation variant of a Java Content Repository is eXo JCR [36]. According to the JCR , eXo JCR supports separate versioning of repository artifacts such as model elements.

The decision, which of the alternative to select, depends on the type of update-strategy in case of changes. If selective updates are desirable, we recommend using versioning for *Model Elements*. If artifacts such as models should be updated as a whole, the alternative of versioning *Models* rather than *Model Elements* should be chosen.

### 4.10 Architectural Decision: Select Change Log Metadata Types

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

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

### 4.11 Architectural Decision: Select Security Metadata Types

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

The first option is to provide *access log metadata*. Hereby, the repository keeps a journal of all significant actions performed by repository requesters on repository resources. Another option is to establish *identity management and authentication*.

Choosing this option means, the repository manages the identity and credentials associated with authorized users and services. Finally, architects can enable authorized users to perform specific actions or to access specific resources by establishing the *authorization and access control* option. The repository provides a mechanism to protect its resources from unauthorized access. In this context, architects can augment a role based access control solution with well-defined authorizations for each role.

### 4.12 Architectural Decision: Select Life Cycle Metadata Types

In this decision, architects have two basic alternatives: They can either *assign a life cycle state to each repository artifact* or implement a *general process model* containing flows of activities. In the latter case, a process engine is needed to drive the execution of activities [8]. When deciding for the first alternative, the *complexity* of the life cycle grows much more than proportional by the number of life cycle states. Thus, if architects intend to use only *basic life cycle actions* such as insert, update, and delete, this alternative is a very effective one.

A known use implementation incorporating life cycle metadata is the ebXML Registry Reference Implementation Project [37]. The exXMLRR project aims at delivering a functionally complete reference implementation for the OASIS ebXML specification [17]. According to the ebXML standard, each RegistryObject instance must have a life cycle status indicator that is assigned by the registry. In contrast, the alternative of using a general process model should be used if there are potentially new actions that will be developed in future. Accordingly, if architects attach a great value on *life cycle scalability*, they should decide in favor of a general life cycle model.

### 4.13 Architectural Decision: Select Association Model

Modeling associations among models and model instances is a commonly addressed problem today. As described in [38] a current problem in process-driven SOAs is to retrieve the relationships between different components, such as which service operations can be invoked from which process activity and which services access which data. Furthermore components that are not depending on any component can be seen as obsolete and thus can be deleted [38]. Another benefit of modeling dependencies between different components is to visualize these dependencies to better support understandability of the models. For this purpose, graphical tools can be designed because the tools are what give value to a repository [39].

As seen in Figure 1 there are two basic alternatives among which architects can choose: As described in our previous work [38], general models can specify associations between certain special-purpose models. In the example, our view-based models of the View-based Data Modeling Framework describe the associations between processes, services, and DAOs [38]. If domain models do not specify associations between them, the repository should handle these associations by defining a general Association Model as specified in the EbXML standard [17]. EbXML's Association Information Model defines classes that enable artifact instances to be associated with each other.

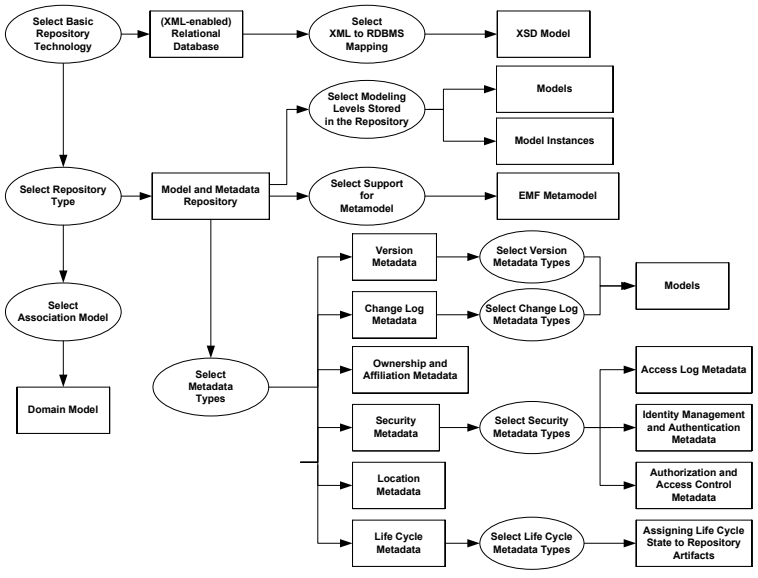



**Fig. 5.** Case Study: Selected Architecture Decisions of the DAO Repository

## 5 Case Study

In this case study we illustrate major design decisions that we made when setting-up our own Data Access Object (DAO) Model Repository. During the design process of the repository we were faced with several fundamental architectural decisions. In this case study we reflect the decisions made to set-up our DAO Model Repository by walking through the reusable architectural decision model presented in the section before (see Figure 5).

Before we walk step by step through our reusable architecture decision model we would like to shortly motivate the use of a Data Access Object (DAO) Model Repository: Developers typically store DAOs in local file systems and concurrent versioning systems, such as CVS or SVN. However, especially as the number of DAOs grows, finding a particular DAO on a concurrent versioning system, in order to reuse the DAO, can become rather time-consuming. Thus, developers need more sophisticated query mechanisms to quickly locate existing database operations in order to increase DAO reuse. The DAO Model Repository supports queries for retrieving desired DAOs by diverse search criteria, such as finding all DAOs accessing a particular database, all DAO operations inserting data into a particular table, or all DAO operations updating a certain column of a table. Moreover, DAO developers are able to query ownership information about a certain DAO and thus look for all DAOs registered by a certain user or department. Furthermore we use a model-driven approach so that DAO developers do not have to deal with various Object Relational Mapping (ORM) technologies. The goal was that developers simply need to generate source code from a chosen model

instance persistently stored in the DAO Model Repository or from a newly defined model respectively to create a DAO for a specific ORM technology.

1. *Select Basic Repository Technology*: When setting-up our DAO repository, we decided in favor of the *(XML-enabled) Relational Database* alternative. Our main decision driver was that RDBMS are very common and hence, we can benefit from tool support. We decided against using a *NXD* system, because our DAO repository models have many associations between them and thus many joins are necessary when querying DAO data. Accordingly, they are the joins in *NXD* storages, that can have longer response times compared to RDBMS. As searching a large number of DAOs could be rather inefficient, for us, a *File System* storage, was out of question.
2. *Select XML to RDBMS Mapping*: We decided in favor of the *XSD Mapping Model* alternative because this approach requires less tables to join and thus results in quicker response times than the *Domain Model* approach.
3. *Select Repository Type*: Our DAO Model Repository should primarily store models, but also to be defined metadata. As a consequence we opted for the *Model and Metadata Repository* alternative.
4. *Select Support for Metamodel*: We chose *EMF* [40] as an explicit metamodel to specify our models of Viewbased Data Modeling Framework [38]. Thus, we can benefit from existing tool support such as openArchitectureWare [31] to generate code from existing model instances.
5. *Select Modeling Levels Stored in the Repository*: Our generated DAO source should be dependent on the specific Object Relational Mapping (ORM) technology such as HIBERNATE [41] or IBATIS [42]. For this purpose our DAO Repository stores technology- and platform-independent *model instances*, that are used for source code generation on the client side. Another requirement was to automatically validate checked-in model instances. In order meet this requirement, we settled for storing *models* in addition to model instances. As they are the repository users that have to generate the source code, they need to integrate required source code generation tools into their development environment. Accordingly, repository users generate *runnable code* by compiling the generated source code. Thus our repository does neither store source code nor runnable code.
6. *Select Metadata Types*: According to the decision of selecting the Model and Metadata Repository as repository type, we decide to add metadata to our repository. In the following we focus on those decisions that are not covered by follow-on decisions: We settled for adding *ownership and affiliation* metadata to being able to efficiently set-up our prototype in medium-sized and large companies. Up-to-now, we do not relate to documentation or source code stored on other repository. Thus we do not store *location metadata* in our repository. As our Repository still is a prototype solution, at the moment, we do not provide *internationalization* metadata.
7. *Select Version Metadata Types*: Our DAO repository requires *versioning* artifacts. However, we wanted to save extra efforts related to *versioning model elements*. Thus, we decided in favor of adding version information metadata to *whole models and model instances*.
8. *Select Change Log Metadata Types*: As this decision is based on the decision of selecting version metadata types, we opted for adding *change log metadata* to *models and model instances* rather than to *model and model instance elements*.

9. *Select Security Metadata Types*: As we intend to provide our repository to industry, we added basic *security* metadata for all three security options illustrated before, namely *access log metadata*, *identity management and authentication metadata* and *authorization and access control metadata*.
10. *Select Life Cycle Metadata Types*: Our repository incorporates a basic *life cycle manager*, that manages basic actions such as insert, update, delete and validate. As we required both a simple solution and the life cycle manager not necessarily to be scalable related to new actions and states, we opted for *assigning a life cycle state to repository artifacts*. We have decided against using a *general process model*, because this solution seems a bit oversized for our prototype repository solution.
11. *Select Association Model*: Our DAO models incorporate relationships between *domain model instances*. Thus, we use our own *domain models* to specify associations between DAO model instances instead of using a *general association model*.

## 6 Related Work

To be able to accomplish this work we were inspired of repositories in general. In [8], Bernstein and Dayal give a fundamental overview of repository technology as well as functional requirements of a repository. Afterwards, we focused on repositories incorporating metadata. A common representative of Metadata Repositories are service repositories that contain metadata about location information such as service bindings according to the Web Service Description Language (WSDL). Here, there exist various standards such as [34], ebXML [17] and related implementations such as the ebXML Repository Reference implementation [37] and the WebSphere Service Registry and Repository that is based on UDDI.

Furthermore we focused on current model repository standards and implementations. As illustrated in this paper, there are many known model repository implementations such as Netbeans MDR [7] that stores models and model instances and Eclipse CDO [6] that stores models, but no XMI model instances. In [43] France et.al.'s interesting approach introduces a development plan for setting up model repositories storing MDD artifacts. In contrast to our paper, the authors of the ReMoDD project in particular focus on the types of interactions that are most useful for repository users. Besides, the ReMoDD project's scope of research does not include storing metadata.

Finally, there are many articles that focus on each of the illustrated decisions for their own. For example, several work [19,20] focus on algorithms of mapping XML model instances to a certain Repository storage type. However, for the best of our knowledge there is no work that connects all these illustrated architecture decisions with each other. In [44] Milanovic et.al. presents an approach of designing and implementing a repository that supports storing and managing of artifacts such as metamodels, models, code, and their metadata. As our approach, the illustrated repository stores metadata such as versioning information. However they do not provide an overview about different types of metadata such as those presented in our work. They exemplary illustrate the design of the BIZYCLE repository architecture without identifying architecture decisions to select different alternatives and options. Instead of involving management issues such as project management and user control, our decisions primarily deal with the question which artifacts should be stored in a repository and how to model the associations between them.

## 7 Conclusion and Outlook

In this paper we introduced a Reusable Architecture Decision Model (RADM) for setting-up Model and Metadata Repositories. These decisions in particular aim at data design for Model and Metadata Repositories. We provided a decision basis for fundamental choices such as selecting a basic repository technology, choosing appropriate repository metadata, and selecting suitable modeling levels of the model information stored in the repository. Our experiences result from developing our own Model Repositories, from researching on other works, discussions with other people involved in repository projects, and applying our RADM in a case study.

Part of our future work could be a more precise evaluation of our decisions based on using quality management methods such as Quality Function Deployment (QFD). QFD could capture the repository's requirements and thus selectively deploying the activities for each decision alternative. Besides specifying reusable architecture decisions for setting-up Model and Metadata Repositories we increasingly concentrate on server-client interactions and repository client tools, that give value to the repositories. Finally, using ontologies for querying repository artifacts could improve the quality of the retrieved result sets.

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

## References

1. Riggio, R., Ursino, D., Kühn, H., Karagiannis, D.: Interoperability in meta-environments: An XMI-based approach. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 77–89. Springer, Heidelberg (2005)
2. Völter, M., Stahl, T.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Chichester (2006)
3. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons, Chichester (2004)
4. Sriplakich, P., Blanc, X., Gervais, M.P.: Supporting transparent model update in distributed case tool integration. In: SAC 2006: Proceedings of the, ACM Symposium on Applied Computing, pp. 1759–1766. ACM, New York (2006)
5. Kramler, G., Kappel, G., Reiter, T., Kapsammer, E., Retschitzegger, W., Schwinger, W.: Towards a semantic infrastructure supporting model-based tool integration. In: GaMMa 2006: Proceedings of the 2006 international workshop on Global integrated model management, pp. 43–46. ACM, New York (2006)
6. Eclipse: Eclipse CDO, `http://wiki.eclipse.org/CDO` (CCopyright2009)
7. NetBeans Community: Metadata repository (MDR), `http://mdr.netbeans.org/` (retrieved January, 2009)
8. Bernstein, P.A., Dayal, U.: An overview of repository technology. In: VLDB 1994: Proceedings of the 20th International Conference on Very Large Data Bases, San Francisco, CA, USA, pp. 705–713. Morgan Kaufmann Publishers Inc., San Francisco (1994)
9. Lloyd, C.M., Lawson, J.R., Hunter, P.J., Nielsen, P.F.: The cellml model repository. Bioinformatics 24(18), 2122–2123 (2008)

10. Taylor, R.N., van der Hoek, A.: Software design and architecture: The once and future focus of software engineering. In: Future of Software Engineering (FOSE 2007), pp. 226–243 (2007)
11. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: Proceedings of the 5th Working IEE/IFP Conference on Software Architecture, WICSA (2005)
12. Zimmermann, O., Zdun, U., Gschwind, T., Leymann, F.: Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In: WICSA 2008: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), Washington, DC, USA, pp. 157–166 (2008)
13. Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 43–58. Springer, Heidelberg (2006)
14. Tyree, J., Ackerman, A.: Architecture decisions: Demystifying architecture. IEEE Software 22(19-27) (2005)
15. Harrison, N., Avgeriou, P., Zdun, U.: Using patterns to capture architectural decisions. IEEE Software, 38–45 (July/August 2007)
16. Zimmermann, O., Gschwind, T., Kuester, J., Leymann, F., Schuster, N.: Reusable architectural decision models for enterprise application development. In: Overhage, S., Szyperski, C., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880, pp. 15–32. Springer, Heidelberg (2008)
17. OASIS/ ebXML Registry Technical Committee: Registry Services Specification v2.0 (December 2001), `http://www.ebxml.org/specs/ebrs2.pdf`
18. Eclipse: Eclipse Modeling Framework Project, `http://www.eclipse.org/modeling/emf/` (retrieved December 2008)
19. Haw, S., Rao, G.R.K.: Query optimization techniques for xml databases. International Journal of Information Technology 2(1), 97–104 (2005)
20. Atay, M., Sun, Y., Liu, D., Lu, S., Fotouhi, F.: Mapping xml data to relational data: A dom-based approach. In: Eighth IASTED International Conference on Internet and Multimedia Systems and Applications, Kauai, pp. 59–64 (2004)
21. Fotsch, D., Speck, A.: XTC – The XML Transformation Coordinator for XML Document Transformation Technologies. In: DEXA 2006: Proceedings of the 17th International Conference on Database and Expert Systems Applications, pp. 507–511. IEEE Computer Society, Los Alamitos (2006)
22. Khan, L., Rao, Y.: A performance evaluation of storing XML data in relational database management systems. In: WIDM 2001: Proceedings of the 3rd international workshop on Web information and data management, pp. 31–38. ACM, New York (2001)
23. Schwede, T., Kopp, J., Guex, N., Peitsch, M.C.: Swiss-model: An automated protein homology-modeling server. Nucleic Acids Res. 31(13), 3381–3385 (2003)
24. BrainML: Neurodatabase construction kit, repository server, `http://brainml.org` (retrieved January, 2009)
25. Nicola, M., van der Linden, B.: Native xml support in db2 universal database. In: VLDB 2005: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment, pp. 1164–1174 (2005)
26. Emadi, M., Rahgozar, M., Ardalan, A., Kazerani, A., Ariyan, M.M.: Approaches and schemes for storing dtd-independent xml data in relational databases. Trans. on Engineering, Computing and Technology 13 (May 2006)
27. Florescu, D., Kossmann, D.: Storing and querying xml data using an rdmbs. IEEE Data Eng. Bull. 22(3), 27–34 (1999)

28. Eessaar, E.: Using metamodeling in order to evaluate data models. In: AIKED 2007: Proceedings of the 6th Conference on 6th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases, Stevens Point, Wisconsin, USA, pp. 181–186. World Scientific and Engineering Academy and Society, WSEAS (2007)
29. Nayak, R., Xia, F.B.: Automatic integration of heterogenous xml-schemas. In: iiWAS (2004)
30. Castano, S., Ferrara, A., Ottathycal, G.S.K., Antonellis, V.D.: A disciplined approach for the integration of heterogeneous xml datasources. In: DEXA 2002: Proceedings of the 13th International Workshop on Database and Expert Systems Applications, pp. 103–110. IEEE Computer Society, Los Alamitos (2002)
31. openArchitectureWare: oaw (August 2002), http://www.openarchitectureware.org
32. Tran, H., Zdun, U., Dustdar, S.: View-based and model-driven approach for reducing the development complexity in process-driven SOA. In: In Abramowicz, W., Maciaszek, L.A. (eds.) Business Process and Services Computing: 1st International Conference on Business Process and Services Computing (BPSC 2007), Leipzig, Germany, September 25-26. LNI, vol. 116, pp. 105–124. GI (2007)
33. AndroMDA: Emf uml2 repository (November 2006), http://galaxy.andromda.org/docs-3.2/andromda-repository-emf-uml2/index.html
34. Clement, L., Hately, A., von Riegen, C., Rogers, T.: UDDI Version 3.0.2, UDDI Spec Technical Committee Draft. (October 2004), http://www.uddi.org/pubs/uddi_v3.htm
35. Nuescheler, D., Piegaze, P.: Other members of the JSR 170 expert group: Content Repository API for Java Technology Specification, Java Specification Request 170 (May 2005), http://www.jcp.org/en/jsr/all
36. eXo: Java content repository (jcr - jsr 170), http://www.exoplatform.org/portal/public/en/product/oemisv (retrieved December, 2008)
37. freebXML: Oasis ebxml registry reference implementation project (July 2007), http://ebxmlrr.sourceforge.net/
38. Mayr, C., Zdun, U., Dustdar, S.: Model-driven integration and management of data access objects in process-driven sOAs. In: Mähönen, P., Pohl, K., Priol, T. (eds.) ServiceWave 2008. LNCS, vol. 5377, pp. 62–73. Springer, Heidelberg (2008)
39. Bernstein, P.A.: Repositories and object oriented databases. In: BTW, pp. 34–46 (1997)
40. Eclipse: Eclipse modeling framework (emf) (2006), http://www.eclipse.org/emf/
41. Hibernate: Hibernate (2006), http://www.hibernate.org
42. Ibatis: Ibatis (2006-2007), http://www.ibatis.org
43. France, R.B., Bieman, J., Cheng, B.H.C.: Repository for model driven development (ReMoDD). In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 311–317. Springer, Heidelberg (2007)
44. Milanovic, N., Kutsche, R.-D., Baum, T., Cartsburg, M., Elmasgünes, H., Pohl, M., Widiker, J.: Model&Metamodel, metadata and document repository for software and data integration. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 416–430. Springer, Heidelberg (2008)