Piotr Habela

Metamodel for Object-Oriented Database Management Systems

Ph.D. Thesis

Submitted to the Scientific Council of the Institute of Computer Science, Polish Academy of Sciences

Advisor:

doc. dr hab. Kazimierz Subieta

Warsaw, November 2002

Abstract. A database metamodel is inherent to every DBMS. Appropriate constructs reflect the declarations of data-definition and data-manipulation languages, and are necessary to implement the internal DBMS mechanisms. Particularly, a metamodel definition provides a base for implementation of database's schema repository. However in object-oriented DBMS these notions are often treated implicitly or (as in case of ODMG standard) suffer from rather ad-hoc approach to their definition. This work is intended to show the importance of providing an explicit database metamodel definition that is both simple and extensible. The main roles of database metamodel have been enumerated. Since different responsibilities of database metamodel together with inherent complexity of object data model can easily lead to unacceptably complicated metamodel definition, the radically simplified, "flattened" form of metamodel structure is proposed. Moreover, the proposed solution assumes the usage of generic means to manipulate database metadata instead of a large set of narrowly specialize operations assumed by the ODMG standard. The prototype implementation of metadata repository is provided to prove the feasibility of the simplified metadata structure. To show the importance of metamodel extensibility, the database schema-based mechanism to support the configuration management of ODBMS application has been implemented over the flattened metamodel structure. A number of additional remarks, concerning future extensions to a database metamodel as well as its capability to accept custom extensions in the context of current challenges for database management systems, have been presented.

CONTENTS

1 Introduction

A definition of a metamodel for an object-oriented database management system (ODBMS) needs to be prepared to serve several different purposes. As the term *meta* suggests, it is a kind of database tool's self-description and in fact one of important roles of a metamodel is to precisely explain the meaning and interrelationships of constructs and features provided by a DBMS. A model developed for a DBMS-based application constitutes a metadata that is stored within the schema. Metamodel, being a model of any such user model, determines a logical structure of the schema repository, and is internally used by the core DBMS mechanisms. The metamodel structure, together with metadata it describes should not be hidden inside a DBMS though. Convenient access to metadata and the ability to extend it with additional task-specific information may be of critical importance e.g. for supporting software configuration management or integration of heterogeneous data sources. Yet another issue is the dynamic nature of a database schema. This requires support for modifying schema, which, together with other actions necessary to maintain a database consistency needs to be described in terms of a metamodel.

The object data model was proposed to better handle the complexity of today's information systems, by providing a richer set of modeling constructs. On the other hand, it led to much more complex metamodel, which needs to define all introduced notions, and more effective approaches to managing the complexity of metadata itself are necessary.

The rapid development of the Internet and the distributed systems technology in general, brings new challenge for database management systems. The interoperability and integration of databases requires means to precisely describe local resources as well as to map data representation to a commonly agreed format. Both these issues are highly relevant to a database metamodel.

1.1 *The role of a metalevel*

This section is intended to provide a closer look at the concept of metamodel as well as to explain the motivation behind explicit definition of such in DBMS.

The meaning of the term “meta” is relative and depends on the entities that are considered as regular objects. Two fundamental kinds of properties the meta-level possess can be distinguished concerning its subordinate entities. The first of them can be called ontological. In order to interpret objects properly (that allows to retrieve some information, modify data and / or verify integrity rules), the description of their structure, interfaces and the meaning of the properties they posses is needed. Since such description would in turn require the definition of the notions used to formulate it, further meta-levels can be necessary.[1] There are several options for terminating such (potentially infinite) hierarchy. The highest level can be defined using the language with formally defined semantics [16]. Alternatively, the ambiguity may be minimized by “loop-backing” the definition of the highest meta-layer and / or by mapping it to concrete implementational structures. The second aspect of meta-level is of operational nature. It is assumed, that a metaobject has knowledge about its subordinate objects and is able to manipulate them. Thus it realizes a usually implicit control of the behavior of regular objects.

As can be seen from the above outline, the metalevel describes issues inherent to all systems. However, its explicitness and accessibility may differ. The extent of the metalevel features determines the application’s ability to discover and modify facts both about available data structures as well as about its own behavior.

Within the programming language domain, the metalevel has to provide the reflective capabilities, intended to support so-called *separation of concerns*.[2] Such features have been classified into two general kinds: *introspective* and *intercessory* capabilities (see e.g. [6]). The former allow examining the structure and functionality of entities available during run time and using that information to dynamically construct requests. These features proved to be essential e.g. for development of generic database browsers or the tools providing transparent persistence for programming language objects (including many Java-based OODBMS, e.g. DB4o [10] or Objectivity for Java [33]). The latter (*intercessory* capabilities) mean the ability to intercept a behavior of

[1] A very intuitive metaphor describing the mapping of this aspect of meta-modeling up the metalevel hierarchy found in [14]: the relations between an object and its class and a class and its metaclass can be compared to the dependencies between a cake and a form used to bake it and between the form and the dice used to produce the form. This understanding of the *meta* term will be assumed in this work in the sense, that the criteria of distinguishing metalevels will be the *instance of* relationships between elements.
[2] A more detailed discussion of this issue is provided in the further chapters.

interest and interwove separately defined routines that can be easily interchanged, and what the original code developer needn't be aware of. Not all flavors of such functionality are qualified as reflective, as the term is more often associated with the former of mentioned kinds. For example the trigger / active rule mechanism is treated as a regular (that is – not reflective) feature. The intercessory capabilities constitute the inexplicable foundation for realizing the separation of concerns postulate though.

It is necessary to note that the lack of support for the abovementioned reflective features from the programming language or database system, significantly complicates the development of software that requires such functionality. Practically, in such case the only choice to get access to metalevel is the use of preprocessor, which complicates the development process.

It is rather intuitive that if a given reflective feature proves to be valuable in general-purpose programming language, its importance for DBMS is at least as big or even greater, because of the shift toward first-category constructs and demands for runtime flexibility that are specific to the DBMS.

1.2 Database metamodel

The domain where the term "metamodel" seems to be used most frequently is conceptual modeling [16]. Nowadays, the best known object-oriented metamodel is probably the Unified Modeling Language (UML) specification [41]. In that case the role of a metamodel is to describe concepts of modeling language and to standardize the well-formedness rules and the metadata interchange formats among tools. On the other hand, for a DBMS the most important aspect of metamodel becomes the way the metadata (and the regular data described by it) are structured and manipulated.

Two important desired properties of DBMS motivate making their metamodel an explicit feature. Those are flexibility and interoperability. Flexibility would require features like:

- support for generic programming through reflection;
- ability to extend the metadata structure with custom constructs;

- intercessory capability, allowing to isolate certain aspects of behavior and to easily change their implementation, perhaps in the spirit of the Aspect Oriented Programming (AOP) [30].

Interoperability would concern integration and collaboration of heterogeneous databases.

Thus the metadata need to be queried not only to discover the structure of the described data, but also to provide some hints concerning their meaning / interpretation. These two roles of metadata are usually referred as appropriately structural metadata and semantic metadata.

As can be seen from the above requirements, the metamodel responsibilities are much broader than just the description of an employed data model. Thus it is practical to define a database metamodel as a description of all those database properties that are not dependent on a particular database state. Particularly, a metamodel implemented in a DBMS formally describes and stores the database schema, together with auxiliary data such as the physical location and organization of database files, optimization information, access rights as well as the integrity and security rules.

Metamodels for relational systems are easy to manage due to the simplicity of the data structures implied by the relational model. In these systems, the metamodel is implemented as a collection of system tables storing entities such as: identifiers and names of relations stored in the database; identifiers and names of attributes (together with identifiers of relations they belong to); and so on. Thus while their features can provide important hint on the required properties of ODBMS metamodel, the designers of the latter have to be prepared to handle the inherent complexity of an object data model. As will be shown in further sections, this complexity could severely limit the usability of ODBMS. Such “metadata management nightmare” (term used in [31]) danger became one of the serious arguments against object databases. Fortunately, this complexity, although inevitable, can be managed effectively, thus making the additional cost of the more expressive data model reasonably low.

1.3 Motivation and scope

The aim of this work was the identification and investigation of various requirements that a design of ODBMS metamodel has to consider. This research was performed in the context of object-oriented DBMS standardization efforts and thus is influenced especially by today’s commercially available technologies and existing standards related to the subject. Taking the ODMG (Object Data Management Group) ODBMS standard as a starting point, the characteristics and drawbacks of existing solutions exemplified by this specification have been discussed. The analysis is intended to provide a possibly complete overview of the issues that need to be addressed. Next, the necessary improvements to the described metamodel definition are proposed. The level of detail of proposed solutions differs, depending on the significance of underlying problem and the level of its awareness expressed in existing literature. The majority suggested improvements are of relatively general nature, as they are not intended to constitute a complete standard-like metamodel proposal. The aspects where the suggested solutions differ substantially from the current state of the art were illustrated by a prototype implementation of a generic schema repository, in order to prove their feasibility and usefulness.

1.4 Results

The result is a set of proposed additions and improvements to existing metamodel solutions, based on the analysis of various ODBMS metamodel roles. The following issues have been addressed:

- A sufficiently precise and unambiguous description of DBMS constructs and supporting features, provided by a metamodel definition;
- Suitability of a metamodel definition for its implementation as a part of a DBMS, guaranteeing good performance and intuitive access;
- Ability of a metamodel definition to evolve as a result of additions or improvements in future versions of base specification or because of custom vendor- or domain-specific extensions;

- The required constructs of a data model, supporting useful modeling abstractions and separation of concerns during design, and their integration into a metamodel definition;
- The support of database schema for software configuration management mechanisms (especially in the context of a database schema evolution);
- Metadata structure openness, allowing for extending it with descriptive information necessary to support the interoperability and integration of distributed databases.

Moreover, a prototype implementation of a generic metadata repository has been implemented. The aim was to prove the feasibility of the least conventional of proposed solutions, namely the radically simplified ("flattened") metamodel structure and the database schema-based utility to manage the software dependency information in the context of software configuration management. The following functionality has been implemented:

- A metamodel definition tool, allowing to develop arbitrary metamodels using the flattened metamodel structure, in terms of metaobject and meta-relationship kinds, meta-attributes describing particular kinds of metaobjects and consistency rules connected with given metaobject or meta-relationship kinds.
- A model management tool for defining models according to previously defined metamodels and for testing their consistency.
- A model browser for convenient viewing of defined models.
- An analysis utility allowing to extract an Objectivity/DB ODBMS schema into the simplified metamodel structure.
- A dependency-tracking code (implemented in the form of a Java *aspect* – see [2]), which is intended to run during the testing phase of an Objectivity/DB application in order to detect all dependencies between the application and the database schema and to store it together with metadata extracted by the abovementioned analysis utility.

1.5 Methods and tools used in this work

The starting point of the research presented in this work is the ODMG (Object Data Management Group) standard [34]. However, due to many drawbacks of that specification, it was not possible to keep the proposed metamodel compliant with the standard. Nevertheless, the proposed solutions are closely related to the original in the sense, that it depends on the established terminology and closely follows the object model known from the mainstream programming languages.

Other important solutions that inspired the proposal presented here are the OMG (Object Management Group) specifications, especially the UML (Unified Modeling Language), MOF (Meta Object Facility) and CORBA (Common Object Request Broker Architecture). The conceptual view of the proposed metamodel has been presented in the form that guarantees it is OMG MOF-compliant, which allowed to avoid some ambiguities concerning the notation and terminology used. All of the abovementioned specifications have been briefly presented in the following chapter.

The prototype implementation of a metadata repository has been realized using Java language and the Objectivity/DB ODBMS to provide persistency. The same environment served as an example for which a metamodel (in the form proposed by this work) has been developed and additional metadata-related features (supporting the software configuration management) have been implemented. The latter were realized using the AOP Java language extension, namely AspectJ [2]. The Objectivity/DB ODBMS and the AOP are described in the following chapter. Moreover, the motivation behind the AOP is presented and its influence on the ODBMS metamodel development is discussed.

Another important solution that influenced this work is the *stack-based approach* to query languages [52]. Although not directly applied here, it substantiates a very important assumption: it is possible to efficiently develop an object-oriented, optimizable [44] query language, seamlessly incorporating a full algorithmic power and following the object relativism principle. Taking into account the commercial success of the SQL, this suggests, that the future research concerning object-oriented DBMSs should assume a more central role of a query language, comparing with the existing ODMG standard or today's commercially available ODBMSs. Particularly, based on the above remarks, in this work it is assumed, that the DBMS metadata should be accessed and manipulated using an object-oriented query language provided with imperative constructs rather than through the general-purpose programming languages' bindings.

1.6 Organization

The remainder of this work provides an overview of the related research and solutions as well as the requirements specific to DBMS metadata management. Based on this context, the proposal of a metamodel architecture and its core features is presented. The prototype implementation using Java and Objectivity/DB ODBMS is described.

The text is organized as follows: chapter 2 provides an overview of the existing standards and tools directly or indirectly related to the issue of database metamodel; chapter 3 enumerates and later presents in detail the desired properties of such metamodel that are confronted with the existing solutions – especially the ODMG standard; chapter 4 describes the proposed solution and in chapter 5 the prototype implementation is presented. Chapter 6 provides some conclusions. Some additional details describing the prototype implementation are provided in appendices B and C.

2 Related research and solutions

This chapter provides an overview of solutions relevant to the topic of object database metamodel definition. As can be seen, the chapter is dominated by the descriptions of standards and some mainstream tools. This is the consequence of the overall orientation of this work, which considers the issue of a metamodel in the context of standardization efforts, and attempts to suggest directions towards broadly acceptable and universal proposal. Thus it is desirable to base it, if not on existing technology and specification (which would be too limiting assumption), then at least on well known concepts and commonly used terminology. Another reason is that there are very few academic papers dealing directly with the issue investigated here. One prominent example of research focused on object metadata management and based on existing standard, namely the OASIS [46] project is briefly mentioned here. Some other papers, which are only partly relevant to the subject, are referenced in the next chapter.

2.1 OMG CORBA – Object Model and Interface Repository

CORBA (Common Object Request Broker Architecture), defined by the OMG (Object Management Group) consortium, remains one of the most prominent and mature standards in the area of middleware for interoperability of distributed systems' elements, although recently it seems to be used less frequently, in favor of EJB and XML technologies [19]. Applying a standardized middleware to realize this task follows the well known rule of computing, saying that many complex design problems can be effectually solved through introduction of an additional level of indirection. In case of distributed systems, the broker mechanism as such additional element, which allows to raise the level of abstraction a developer deals with, making the design independent of the following factors:

- hardware and operation system platforms of distributed system's constituents;
- server-object location;
- client's and server's implementation languages and their internal representation.[3]

[3] Note that in case of CORBA objects the terms *client* and *server* are relative to particular interaction. A given object can act as a server, providing functionality to its clients, while at the same time being dependent on a functionality of some other interfaces, thus acting as a client.

This allows CORBA-based solutions to successfully address the following tasks:

- integration of heterogeneous systems;
- easy evolution of deployment configuration, including scalability and load-balancing;
- interoperability of different broker implementations thanks to the use of common protocol (IIOP) built on top of TCP/IP;
- ability of mutually independent development of the client and the server elements intended to cooperate in a distributed system. The only "common denominator" of both parts remains an abstract, programming language-independent interface definition.

The lookup of object references, message passing, security, consistency and a number of other issues are supported by Common Object Services, also defined as parts of CORBA standard.

The IDL object model

What is the most important, the standard has established an architecture for cooperation of heterogeneous systems at the level of language-neutral object's interfaces. Those interface declarations serve later to generate client's and server's code elements in the chosen implementation language (according to standard-defined language mappings). Those interfaces are defined using Interface Definition Language (IDL), which, in order to achieve a better conceptualization is based on an object data model.

The IDL model, although programming language-neutral, is based on the main constructs of C++ language. This is of course advantageous, as it makes the mapping between IDL and today's mainstream object oriented languages quite straightforward.

The most important IDL concept is *interface*, which can specify features mapped into externally available properties of an object of the class that implements it. Inheritance among interface definitions is supported. The properties defined by an interface can be *operations* and *attributes* and the types used to define their signatures can be of following kinds:[4]

- primitive types, like *float, double, byte, string* etc., mostly reusing the keywords known from the C++, whose standard-defined mapping leads usually to appropriate primitive types of particular programming language binding;
- object types, described by *interfaces*, which are mapped into class definition in all languages supporting such a construct;
- sequences of abovementioned types – parameterized type, which e.g. in Java is mapped into a static (typed) array;
- structures (using the *struct* keyword) – a constructor for record-like structures borrowed from the C++ language; in Java it is mapped into a class;[5]
- a generic *any* type, able to accommodate value of every IDL-defined type; it is useful e.g. in generic programming interfaces.

Operation's signature written in IDL can also specify *exceptions* that can be raised during execution of a given operation. Exceptions are also defined in IDL, where they can be specified together with their attributes, to additionally describe the exceptional situation if needed.

It is necessary to note, that all the access to an object defined by the abovementioned declarations is realized through the remote invocation of operations. That is, only the operation calls, their parameters, non-object parameter values, exceptions and object references are passed by a broker between client and server. This means, that each attribute defined as a field in an interface specification is in fact realized by a pair of (overloaded) operations with the same name as the attribute: one (returning value of appropriate type) to read it and the other (with appropriate input parameter) to modify it. The difference is not purely technical: considering the cost of remote invocations and object reference passing, it can significantly impact the detailed design [48]. Assume an example where one, having access to remote object of type *Department* wants to change salary of one of its *Employee*s named "Smith". If the scenario was the following: 1) getting all references to *Employee* objects managed by a

[4] Module declarations are also supported to provide namespaces for type declarations.
[5] *Unions* and *arrays* are also supported.

given department; 2) invoking a *name()* operation on each of returned object references and comparing it with “Smith”; 3) updating the state of the found object, it would mean a significant overhead in terms of costly remote invocations and reference passing.[6] The more pragmatic solution would be following: the *Department* interface provides operations *getEmployeeNames()*, returning a sequence of string values and *getEmployee(string name)*, returning a reference to selected object. This illustrates, that requirements imposed by a distributed system may contradict some rules of object-oriented design, when encapsulation, low coupling and more identity-oriented programming style are considered.

To sum up, the Interface Definition Language provides basic object-oriented constructs, whose granularity, meaning and even syntax closely follow appropriate declarative elements of the mainstream object-oriented programming languages.

Dynamic Invocation Interface and Dynamic Skeleton Interface

The static invocation model mentioned above assumes that the code responsible for invoking and passing the requests to objects of particular interfaces is compiled into applications. Another mechanism, called Dynamic Invocation Interface (DII), allows for construction and execution of request without static (that is, compile-time) knowledge of accessed interfaces. This is provided through the following features, outlined here in the sequence they are usually used:

- The *create_request(..)* operation is declared in standard-defined root interface *Object*, to create one-use request object connected with a given instance. This operation specifies the name of the target’s dynamically called operation and (optionally) a sequence of provided parameters (declared using the *any* type). The result is a *Request* type object, whose interface is described below.
- The *add_argument(..)* operation of *Request* object can be optionally used, if parameters had not been provided at the request creation. All expected arguments need to be provided in a proper sequence.
- The *invoke(..)* operation performs the created request, making available the return value of the invoked operation (if applicable), which is provided as an output

[6] More recent “objects by value” specification can be helpful in certain conditions to avoid such overheads.

parameter of the *create_request(..)* operation. The *delete()* operation removes the used request object.

- The *send(..)* and *get_response(..)* operation pair can be alternatively used for deferred synchronous calls. The latter can be used (if applicable) to return the result of the request (and to check for execution errors). The *poll_response()* operation allows to check, if the request has already been completed.
- Additional operations (*sendp(..)* and *prepare(..)*) can be used for preparation of persistent requests (allowing for asynchronous calls), as well as for using a callback-style asynchronous calls (*sendc(..)* operation).

The obviously missing element is the reflective capability that would allow to extract metadata used later to create a dynamic request. This issue will be described in the following sub-section.

The Dynamic Skeleton Interface (DSI) is a solution analogous to DII, but located on the side of interface implementation. Appropriate object can be dynamically registered as providing the implementation of particular interface. Such an object can then respond to requests using the information provided within a *ServerRequest* object, whose properties include the following:

- A read-only operation identifier;
- A list of parameters, allowing to read input parameters as well as to set the values of the output parameters;
- Operations to set the result value or to raise an exception.

Interface Repository

The Interface Repository provides for the storage, distribution, and management of a collection of related objects' interface definitions [35].

If the definition of a given object is not compiled into an application, in order to access such object it is necessary to extract appropriate interface specification. Apart from generic programming (as suggested above), such information may be necessary in a number of cases, e.g. to support inter-ORB object passing. The Interface Repository (IR) provides functionality to retrieve such information, that is, the specification analogous to the one provided with IDL declarations of registered interfaces. The repository provides operation to directly define new interfaces within it. Alternative ways of storing such definition include compilation of an IDL file or copying interface definition from another repository.

With presence of a consistent Interface Repository it is possible to invoke on an object reference the reflective operation *get_interface()*. Similarly, like abovementioned *create_request(..)* operation, it is defined within the *Object* interface and thus available for all CORBA objects.

Each interface definition has assigned its repository identifier, which allows to maintain the identity of such metadata in presence of multiple repositories. Version number of an interface is also stored, although the definition versioning is not supported by any additional mechanism nor semantics [35]. A particular interface definition can be located in three ways:

- Directly from the ORB (e.g. through the mentioned *get_interface()* operation on *Object*);
- By navigation through the module name spaces (that is, by name);
- By lookup of a specific identifier (that is, by an ID, which may be useful to find a definition corresponding to another) [35].

With presence of full metadata manipulation functionality, the consistency of the repository presents a hard problem. Indeed, only the most obvious inconsistencies (like e.g. name conflict within one interface definition) can be immediately detected and reported. Thus, the flexibility allowing different means to directly update metadata is provided at the cost of leaving the consistency of a repository practically unprotected.

Including recent standard's metadata extensions towards the component model, the Interface Repository specification now consists of nearly 50 interfaces, which constitutes a really complex structure to be queried and manipulated by programmer. Moreover, it is assumed that further extensions (both defined by future standard's versions as well as custom, domain- or tool-specific extensions of standard defined interfaces), would be introduced by specialization of existing definitions.[7]

[7] As explicitly stated in the standard specification, the IR is intended to store additional interface-related information like e.g. debugging information, libraries of related connectivity code etc. [35].

Despite significant complexity the community seems to accept the solution, as being a natural consequence of overall standard's assumption, to provide a possibly direct support for a number of existing mainstream general-purpose programming languages.

However, the programming against the Interface Repository is commonly perceived being very difficult or at least inconvenient. Since in case of database schema the analogous structures would constitute the core feature instead of auxiliary service, following the same style in construction of a database metamodel would be controversial.

2.2 OMG UML

The Unified Modeling Language (UML) provides a graphical notation for visualizing, specifying, constructing, and documenting the artifacts created at different phases a software development process [5],[41]. The language was defined as a unification of three most popular object-oriented software development methodologies (Booch Method – by Grady Booch, OOSE – by Ivar Jacobson and OMT – by James Rumbaugh) and soon accepted as a standard modeling language by the OMG, which allowed to overcome chaos that previously took place within the object-oriented modeling methods area. Moreover, the fact that UML, in contrast its predecessors, does not prescribe a particular development process, made it easier for this proposal to succeed. The UML is now considered as a dominant notation for software systems' modeling and design.

The language defines a rich number of notions together with graphical notation elements used to visualize them. The following kinds of diagrams are supported:

- **Use cases** diagrams, used to express functionality of a given system or subsystem, together with external entities (called *actors*) that either expect particular functionality or contribute to it.
- **Class** diagrams, used to model the structure of a system under design. Those diagrams constitute the central element of practically every design (including even business modeling), thus it is not surprising that this part of the language has been most precisely described. The assumed semantics of classes is strongly inspired by

Java and C++ language solutions. While this solution makes the UML well prepared for creating detailed design of software written with those languages, at the same time it may be perceived as a factor limiting a conceptual modeling as well as a design for less common implementation platforms.

- **Interaction** diagrams exist in two forms: **sequence** diagrams and **collaboration** diagrams, both intended to show (from different viewpoints), how systems behavior is realized in terms of object interactions.
- **State** diagrams allow modeling a behavior of an object of a given class or of a whole system from the point of view of the lifecycle of such object or system.
- **Activity** diagrams that so far seem to be rather loosely connected with the rest of underlying model, serve as a general mean of visual description of e.g. method's algorithm or a business process.
- **Component** and **deployment** diagrams allow to illustrate appropriately the structure of implemented software and its target location within the physical deployment configuration.

Moreover, to specify additional constraints not expressible by standard graphical notation elements, a precise declarative (and state-preserving) constraint language named OCL (Object Constraint Language) has been introduced.

As it became clear, it is practically impossible to foresee and define all constructs and properties that could be required for such a wide area of application, the special language extensibility features have been defined. Thus the UML metamodel provides three kinds of supporting features that can be used to extend the metamodel:

- **Constraints** allow to specify additional conditions, which could not be covered by applying standard constructs (e.g. available in UML class diagrams).
- **Tagged values** are the tag-value pairs that can be added to a model element to provide additional information (e.g. author, version etc.).
- **Stereotypes** are in fact the only element kind capable to extend the predefined metamodel. Stereotype is a mean of meta-classification, and in its simplest form it just marks a given instance of the metamodel element, e.g. class (stereotype definition requires specifying exactly one metamodel element as its base) to which

(including its descendants) a given stereotype is applicable. In its more sophisticated forms stereotype may extend metadata connected with a given element, by declaring *tag values*, that become effectively additional attributes describing model element. Similarly, the stereotype definition may contain constraints that would be imposed on every instance of a metamodel element the given stereotype is assigned to.

Stereotypes defined to support particular problem domain (e.g. software-development methodology or detailed design for particular implementation platform) may be provided as UML *profiles*, defined outside of the language's specification. The concept is rather controversial, as it is defined in the UML metamodel together with the notions the stereotypes' instances are supposed to extend/redefine. Thus every particular instance of stereotype appears one meta-level lower than the notion it redefines. Anyway, the presence of the notion clearly indicates the need to provide means for lightweight extensions of standardized metamodels.

From the point of view of this work, the UML is especially interesting as the source of the most popular object-oriented metamodel, which provides quite a useful and expressive (although informal) definition of the meaning of the introduced language constructs. It is doubtful as to whether such a metamodel is a full description of UML semantics. This definitional style suffers from the *ignotum per ignotum* logical flaw (concepts are defined through undefined concepts; definitions have cycles but they are not recursive). The metamodel bears informal semantics through commonly understood natural language tokens and a semi-formal language. The formal data semantics of class diagrams can be expressed through a definition of the set of valid data (database) states and by mapping every UML class schema into a subset of the states [53]. Semantics of method specifications requires other formal approaches, e.g. the denotational model. Such a formal approach would radically reduce ambiguities concerning UML; however, due to the rich structure and variety of UML diagrams, the formal semantics presents a hard problem. Instead of using formal semantics, the UML metamodel presents an abstract syntax of data description statements, and various dependencies and constraints among introduced concepts.

Apart from its descriptive role, the UML metamodel allows for definition and verification of consistency rules among the abovementioned different views of a modeled system. It also serves as a base for definition a metadata interchange format, which is already standardized as a XML-based solution named XMI (XML Metadata Interchange) [42].

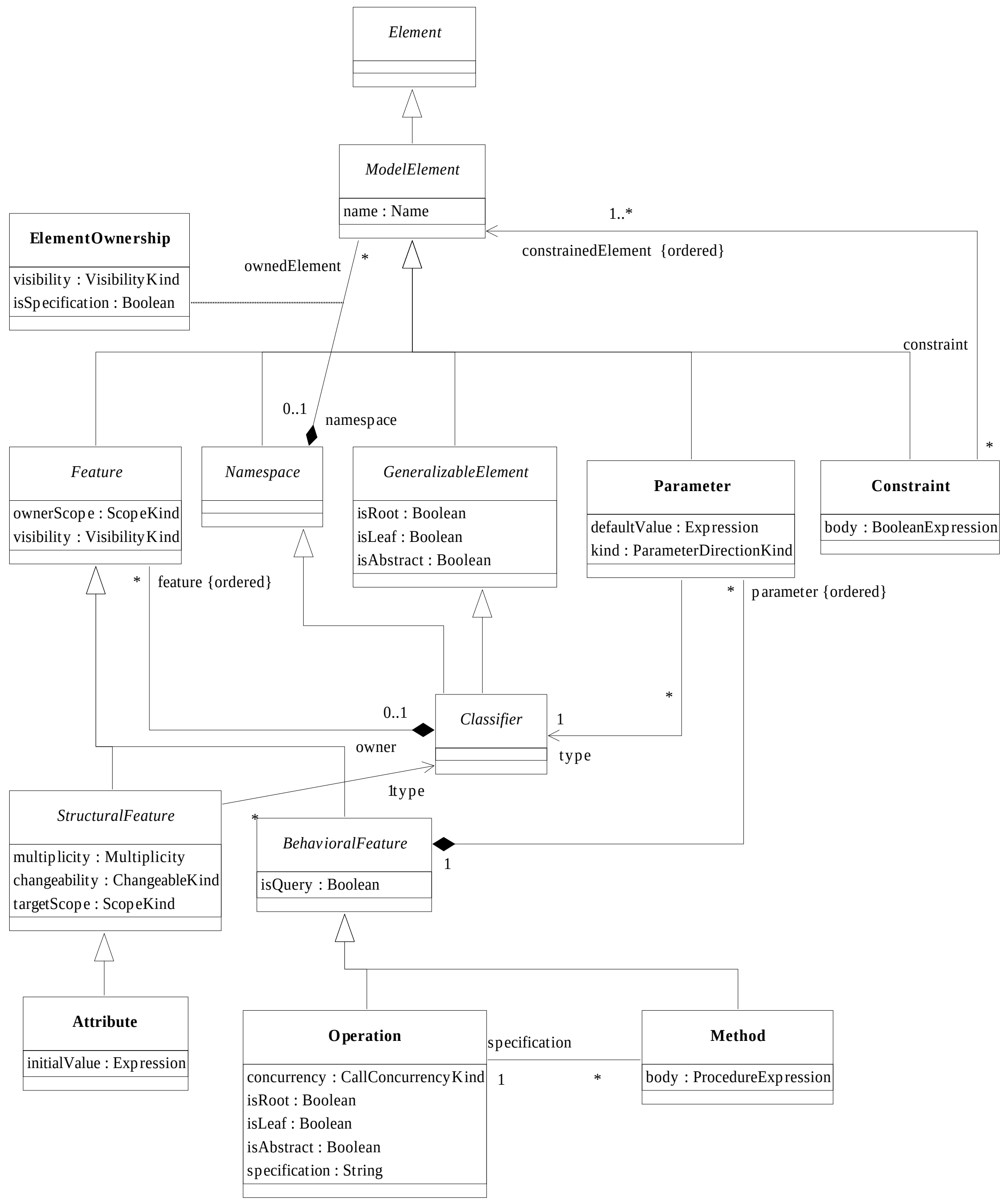


Fig. 1. A fragment (ca 20%) of UML metamodel, including the core language elements

The UML metamodel is fairly large, which is not a surprise concerning the multitude of different views supported by this language. Moreover, additional complexity results from the attempt to make the definition highly generic. Note for example, that the *Class* concept, being a direct specialization of *Classifier* (shown in Fig. 1), has a hierarchy of four more general notions above it (that is, *Classifier, Generalizable Element*, *Model Element* and *Element*) and many of its properties are defined by them. Since a number of such features seem to be defined rather too high, the combinations of allowed properties need to be restricted in the subclasses, leading to extensive usage of additional constraints (formulated in OCL).

2.3 Meta Object Facility

Meta Object Facility (MOF) is an OMG specification, which "*defines an abstract language and a framework for specifying, constructing, and managing technology neutral metamodels*" [39]. It is thus intended to provide a common base in term of which other metamodels like UML, IDL, CWM and others (not necessarily limited to OMG standards) could be uniformly described. The effort is not limited to just a common conceptual base for meta-modeling, as the definition provides also a framework to implement arbitrary metadata repositories.

To achieve this, another, higher meta-level was necessary, which led to application of the common four-layer approach to meta-modeling. Fig. 2, based on similar schema in [39], illustrates resulting metalevel hierarchy.

Traditionally, a developer is able to define the model layer (M1 level in Fig. 2) for a given problem domain, to determine the structure, constraints and behavior of a data (M0) to be stored in considered system. Four-layer metadata architecture provides the additional flexibility, by allowing to define metamodel (M2) elements to be used during modeling. Metamodels are defined using constructs of meta-metamodel (M3), which constitutes an immutable definition defined by such modeling framework and hardwired into a code of tools that support it.

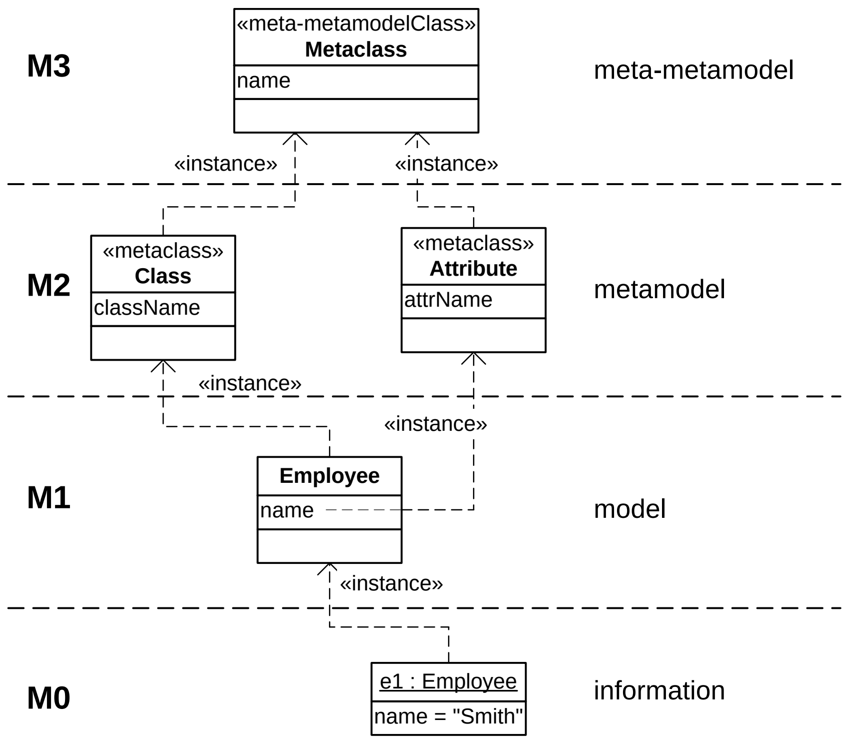


Fig. 2. Illustration of the four-layer metadata architecture (based on fig. 2-1 from [39])

The MOF defines an object-oriented modeling framework, which was intentionally made very similar to the UML metamodel. It is based of four main modeling concepts [39]:

- **Classes**, which can be used to define metaobjects;
- **Associations**, limited to binary ones, to model relationships between metaobjects;
- **Datatypes**, to model other data (including primitive types);
- **Packages**, to modularize the models.

Moreover, similarly like in the UML, the *constraint* notion exists. However, the standard does not prescribe any particular constraint definition language, nor it defines the mechanisms or scenarios to enforce them.

Of course, compared to UML, the MOF metamodel is significantly simplified for at least two reasons. Firstly, as it is limited to structural notions and secondly, because of the intent to directly implement those modeling constructs, which would be impractical in case of sophisticated concepts like e.g. n-ary associations. To realize it, the specification defines a standard mapping between models created using MOF constructs and implementation platform like e.g. CORBA IDL interfaces (optionally also accompanied by a generated repository server code).

In other words, using standard constructs of MOF (M3) one can generate interfaces (M2) to manipulate metadata (M1) on a given platform. It is of course possible to use the framework analogously at the lower level, to deal with regular data (M0), but the intended usage is the development of metamodels to build a universal repository (instead of using the MOF as a ultimate modeling language).

The genericity of the specification is additionally strengthened by the reflective interfaces, allowing for proper interpretation of metadata without previous static knowledge of its metamodel.

The following features of the four-layer architecture are suggested within the specification as making it advantageous:

- Openness; that is, ability to support any possible modeling paradigm;
- Possibility to explicitly define relationships (or mappings) between different kinds of metadata;
- Incremental addition of new metamodels or their elements is possible;
- The common meta-metamodel constitutes a base that allows to interchange different models and metamodels between parties.

The issues like model-to-model transformation mechanisms or a specification of modeling notations for metamodels developed with the MOF are expected to be addressed in future versions of the specification.

Two related specifications establish very important connections between the MOF and other standards or tools dealing with metadata. The first of them is the UML profile for MOF (part of EDOC specification [37]), providing a standard way of visual design of metamodels defined in terms of MOF constructs. The mapping is two-way, thus it also allows to view the MOF-compliant models with UML. In particular, following those specifications can provide a commonly understood base for formulating proposals of different metamodel specifications.[8] The second of mentioned specifications, even

[8] Indeed, the most of conceptual diagrams presented in the following parts of this work as UML class diagrams can be considered to be MOF-compliant.

more important, is the XMI specification, which describes a standard way of expressing MOF-compliant metamodels and metadata in the W3C's XML [62] format. This provides a widely-accepted format for the interchange of different types of metadata e.g. among modeling tools (thanks to the fact that UML metamodel is also defined in terms of the MOF) or different type of data repositories or data-analysis tools.

2.4 Common Warehouse Metamodel

The OMG CWM (Common Warehouse Metamodel) [36] standard addresses the issue of metadata interchange in the area of data warehousing. It provides a metamodel definition specialized for this problem domain, although being independent on any specific data warehousing implementation. With a wide scope assumed by the specification (description of a whole data warehouse system) it is intended to establish generic data warehouse architecture [36].

The base of this specification is the *Object Model*, which has been created as a subset of the UML metamodel. Thus all the standards discussed so far share the same object model as a base for additional metamodel elements, specific for the application domain of a given standard. Another package, named *Foundations*, provides the rest of basic metamodel elements, which in contrast to the *Object Model* package are specific to the data warehousing domain. It consists of concepts describing business information (parties, locations, contacts, documents etc.), datatypes (extending those covered by the *Object Model*), expression representation, base concepts for indexing the data, software deployment (sites, machines, components etc.) and mapping between data types (from different systems).

Additional packages, based on the abovementioned ones, concern among others the following issues:

- Data resources (with support for different data models used to store them, like record, relational, object-oriented and others);
- Data analysis (including data transformation and visualization, OLAP, data mining).
- Warehouse management, including description of operations performed on data warehouse system.

The details of the standard are irrelevant to this work. However, it is outlined here as an example of a domain-specific metamodel definition, based on the standard OMG's modeling framework. Technically, the integration with UML and MOF is achieved by defining the CWM in terms of MOF meta-metamodel. Particularly, this allows to apply the XMI mapping to a warehouse description, which allows for [36]:

- Transformation of the CWM metamodel into XML's Document Type Definition DTD;
- Transferring warehouse metadata in the form of XML documents conforming to the abovementioned DTD;
- Providing the CWM metamodel itself in a form of a XML document (based on DTD defined by the MOF standard) in order to use it in generic MOF-compliant repositories.

2.5 *Model Driven Architecture*

The most recent significant initiative of the OMG, named Model Driven Architecture (MDA) [38], takes advantage from the strong integration among abovementioned standards. It is intended to support rising a level of abstraction used in software development and integration. More precisely, it provides a UML-based modeling framework assuming a clean separation between business logic model and the design elements dependent on infrastructure (programming languages, middleware etc.) selected to deploy it. This is motivated both by the need to manage the complexity of system's specification, as well as by the observation that an enterprise has to deal with a number of different deployment platforms, which additionally (in a longer perspective) are definitely a subject of change. Moreover, a very popular postulate of reuse, raised here to the level of conceptual model instead of implementational artifacts, also comes into play.

The specification distinguishes the following main levels of abstraction:

- Computation-independent business model (also called a domain model);
- Platform Independent software Model (PIM);
- Platform Specific software Model (PSM);

- Implementation.

Although analogous mappings exist between all those levels, the specification is focused on PIM and PSM, which are the most important for the development effort and at the same time precise enough to consider support for their automatic mappings.

The most important concept used by the MDA is the cross-model *refinement* correspondence that holds between a more abstract base model and another model of the same system, which adds details determined by a design decision or perhaps predefined mapping (called *refinement pattern*). A transformation of different nature occurs during *zooming-in* or *zooming-out*, which means changing the granularity of details presented by a model. This mechanism is related with the model *packages* composition. A yet another concept that explains differences between models is a *viewpoint*, specifying an abstraction criteria according to which a given model is built. A *viewpoint correspondence* holds between two models created according to the same viewpoint, which is necessary to determine e.g. when specifying two systems to be integrated.

The above overview may lead to the question, if the MDA is not just a kind of methodology, promoting design and integration good practices and depending on the UML and related specifications. In fact, a more distinct contribution of MDA may be expected from specializing existing modeling framework towards support of specific PIMs and PSMs. Particularly, this would mean:

- UML profiles for PIMs. At this level the number of necessary profiles is assumed to be very small, as they need to indicate only a general domain of developed system (e.g. enterprise computing, real-time or other) in a platform independent fashion.

- So-called *pervasive services* definitions. Those are specifications of the most important services required by a distributed object environment (of course inspired by CORBA's Common Object Services specifications), defined on a higher level of abstraction to make them platform-independent. The specifications of *pervasive services* will need to be backed by platform-specific definitions for all of the middleware platforms supported by the standard [38].

- UML profiles for PSMs. Those would define (through stereotyping of appropriate UML modeling notions) the constructs available in particular implementation tool or platform. For example, the profile for CORBA's PSM may stereotype the UML's *class* concept to distinguish its IDL constructs like *interface*, *valuetype*, *struct* etc.

- PIM to PSM mapping patterns or guidelines. Whenever it is feasible, the transition from PIM to PSM should be supported by standard mapping, allowing to automatize such development step. If a PIM is not precise enough to determine optimum mapping, the *annotations*[9] accompanying such PIM may be taken into account. If there is still a number of possible choices, guidelines or patterns referencing a given mapping case may support developer in manual choice of an optimum solution.
- PSM to PIM mappings to support reverse engineering.

Making the last two of mentioned specification elements precise enough to be a subject of tool support may be feasible only in a limited scope. On one hand a rich repertoire of implementation constructs may be available for a given PSM. On the other hand, the modeling language of PIM may be too limited. For example current UML version allows for pretty precise specification of structural aspects, thus making it possible to map this aspect of specification into a PSM. However the lack of adequate UML description of behavioral semantics is a limiting factor, whose removal would be very important for future evolution of the language [20]. The MDA may also require the UML to improve its ability to describe the *refinement* relationships as well as the *zoom-in* and *zoom-out* model capabilities.

2.6 ODMG

The Object Data Management Group (ODMG) was created in the 1991 by a number of the OMG member companies, together representing almost the entire object-oriented database systems' industry. The aim was to strengthen the market position of ODBMSs, by establishing a set of de-facto standards, allowing for the portability of database applications between different ODBMSs. Within the next ten years, the ODMG issued four versions of object-oriented database standard, which, comparing to its relational database counterpart, the SQL, failed to gain a significant popularity, at least it the commercial world.

[9] Annotations are assumed to be platform independent. An example of important information brought by annotation, provided in [38] is the distinction of conceptual-model classes into describing an entity or a process.

As explicitly stated, the main assumption of the standard was to borrow as much as possible from existing specifications, namely the OMG CORBA and SQL, and to align its definition with the properties of existing products. This led to an architecture characterized by the following design decisions:

- A database is provided as a persistence feature added to a general-purpose object-oriented programming language, which plays a role of database's data manipulation language and uses standardized API to explicitly invoke DBMS's operations.

- The assumed object model is programming language-independent, although closely follows the common elements from object models of the supported languages. The most recent version of the ODMG standard supports C++, Smalltalk and Java, defining appropriate APIs and language mappings from the abstract Object Model. The way of manipulating objects is language specific. E.g. in Java language, persistence through reachibility and garbage-collection concerning database objects are assumed.

- Database schema can be defined using Object Definition Language (ODL), defined as a superset of the CORBA's IDL. The most important enhancements concern bidirectional, integrity-maintaining binary relationships between objects and collections (*set*, *bag*, *list* and *map*), being the primary mean to organize database's data structure.

- Database does not store objects' behavior. It can be specified only by methods in applications' code, and is performed exclusively on the client's side. This kind of architecture is usually called a *passive server*.

- The syntax of the standard's query language, named OQL (Object Query Language) is inspired by SQL, although the similarities are rather superficial (especially, considering the notion of object's identity). In contrast to the SQL, the OQL lacks imperative constructs, although it allows data manipulation through invoking methods (which may have side effect), by selecting them within the "select" clause. Since the features of the OQL make it insufficient for writing complete applications, the OQL statements (similarly like in case of SQL) are supposed to be embedded within a programming language code. In effect the so called "impedance mismatch" problem of embedded SQL occurs also in case of the OQL.

Taking into account the strong connection between database and programming language, the complexity of the OQL and ambiguities of its definition, the role of the query language in the above architecture unfortunately becomes rather secondary, which is confirmed by a very limited support for the OQL provided by commercial ODBMS. This may be perceived as a step backwards compared to relational databases, since using solely programming language to manipulate database objects is a low-level and not a very productive approach. This seems to be the most serious and fundamental source of controversy around the ODMG specification. Another reason is a generally low technical quality of the specification, which is exemplified e.g. by issues discussed in the next chapter.

The ODMG metamodel, is defined through a collection of ODL interfaces, which are intended to provide access to an ODBMS schema repository, organized analogously to CORBA's Interface Repository, which has been the pattern for this definition. As can be seen from the fragment presented in Fig. 3, the structure of the metamodel is very complex, and despite the total number of interfaces (31) is smaller compared to the latest version of Interface Repository specification [35], its usage seem to be even more complicated. The ODMG metamodel specification still seems to be immature, and the consequences of providing some of its features (e.g. metadata-updating operations) have not been addressed so far. This is additionally confirmed by the fact that e.g. the latest ODMG Java binding specification has not accommodated those interfaces at all, while the C++ binding supports them only in their read-only part.

Since the ODMG standard metamodel remains the most relevant specification to the topic of this work, some of its issues will be further discussed in the following chapters.

Since the publication (in 2000) of version 3.0 of the specification [34], the standard seems to be stagnant. Currently, the only actively developed and implemented sub-area of that standard concerns support for Java language and took the form of the Java Data Objects (JDO) specification, influenced in some extent by the ODMG Java binding.

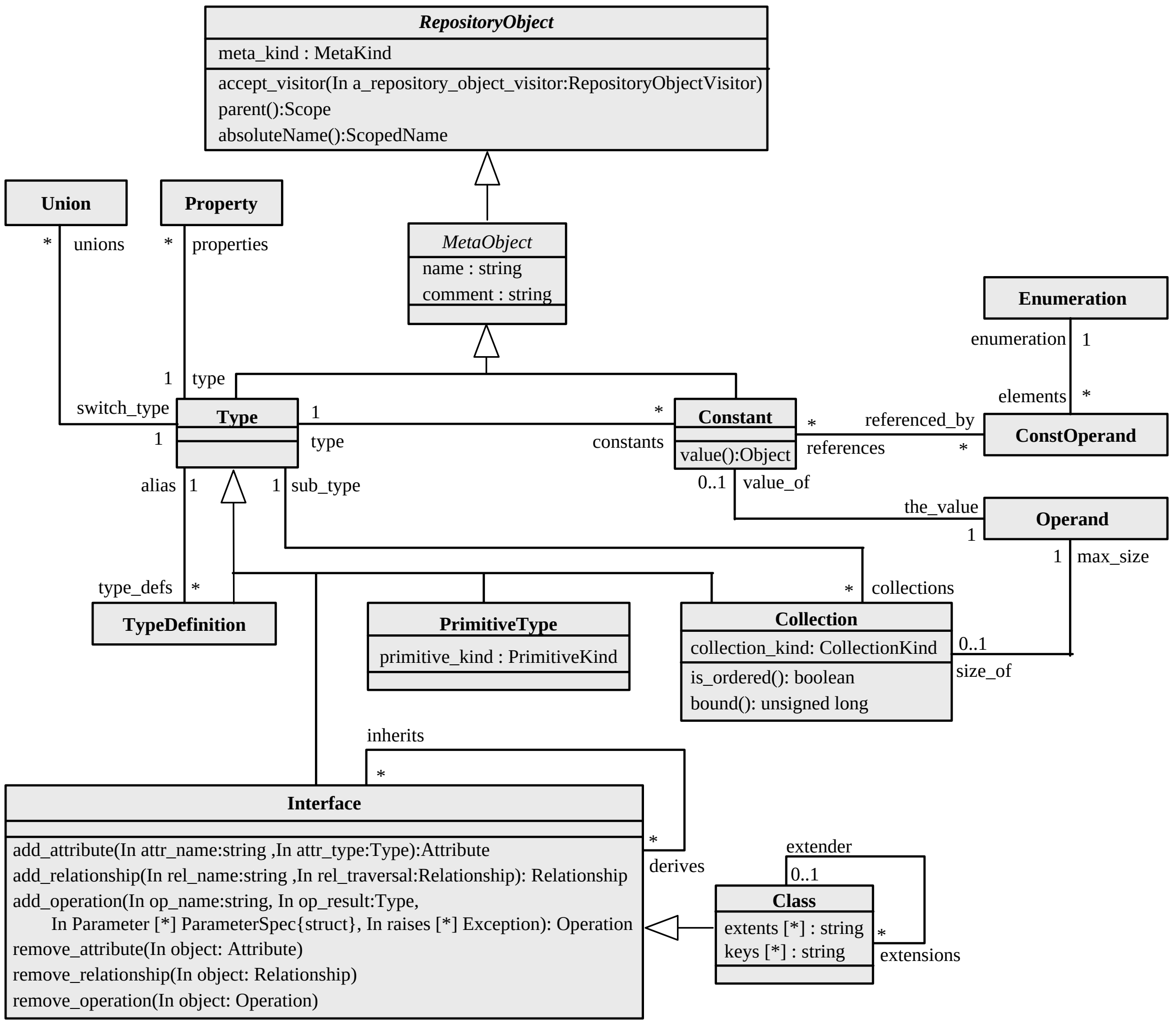

RepositoryObject
meta_kind : MetaKind
accept_visitor(In a_repository_object_visitor:RepositoryObjectVisitor)
parent():Scope
absoluteName():ScopedName
Union
Property
MetaObject
name : string
comment : string
Enumeration
* unions
* properties
enumeration 1
elements *
1 type
switch_type
Type
1
* Constant
value():Object
* referenced_by
ConstOperand
1
type
constants
references *
alias 1
1 sub_type
0..1 value_of
the_value
Operand
1
1 max_size
type_defs *
* collections
TypeDefinition
PrimitiveType
primitive_kind : PrimitiveKind
Collection
collection_kind: CollectionKind
is_ordered(): boolean
bound(): unsigned long
0..1
size_of
inherits
*
Interface
add_attribute(In attr_name:string ,In attr_type:Type):Attribute
add_relationship(In rel_name:string ,In rel_traversal:Relationship): Relationship
add_operation(In op_name:string, In op_result:Type,
In Parameter [*] ParameterSpec{struct}, In raises [*] Exception): Operation
remove_attribute(In object: Attribute)
remove_relationship(In object: Relationship)
remove_operation(In object: Operation)
*
derives
extender
0..1
Class
extents [*] : string
keys [*] : string
*
extensions

Fig. 3. A segment (ca.25%) of the ODMG metamodel

The JDO [27], being maintained by Sun Microsystems Inc., is intended to provide a uniform programmer’s interface to manipulate persistent objects from different data sources, treating them as common Java objects. The scope of this standard is much more modest compared to the ODMG, which is reflected e.g. in the lack of a query language specification. In this area, the standard provides only an interface named *Query*, which allows for simple extent or collection filtering, based on predicates defined over its objects’ state. The queries can be paremetrized, but on the other hand, do not support using method calls in the predicate body.

The JDO does not define special support for metadata management. Its features in this area are almost identical to regular Java’s reflection mechanism, and are provided through a special interface, allowing to avoid using reflection during runtime.

2.7 The OASIS project

The aim of the OASIS (*ODMG Architectures for the Specification of Interoperable Systems*) project [46] was the construction of a federated healthcare system, based on the ODMG as a canonical data model. This kind of application is especially interesting because of the need to integrate heterogeneous data sources, which depends heavily on accessing and manipulation of metadata. The resulting publications provide a number of interesting remarks on the required features of ODBMS architecture in general and its metadata-related features in particular. It is worth to mention the following issues:

- The approach used to integrate systems into a federation extensively used a view mechanism to define data sources in the form suitable for exporting from local systems as well as to integrate them into a federated schema. This requires a powerful view mechanism, including appropriate metamodel constructs, since the virtual classes (including inheritance graphs), attributes and relationships created by a view definition need to be explicitly represented in a schema repository in a way analogous to metadata of a base schema.
- An integration of distributed databases makes the *passive sever* architecture, prohibiting the behavior sharing, inappropriate for several reasons. Firstly, it jeopardizes consistency, since there may be a number of implementations realizing logically the same behavior on the same data. Secondly, if the behavior resides only on the client's side, the federated view definition cannot access it, thus it has to be limited to dealing with static objects' properties only. Thirdly, in case of multimedia object extraction, inability to select the searched fragment (e.g. a scene in a long video sequence) on the server side may result in a big data transfer overhead [28]. Another drawback of the ODMG specification discovered in the context of a federated system construction was the lack of standardized event notification mechanism, essential for propagating updates throughout the federation.
- The integration of heterogeneous databases (mapped for this need into a chosen canonical data model) requires extracting the metadata that describe the organization of provided resources. Experience showed that despite standardization

of API, the portability of generic applications using reflection mechanism among different DBMS is not possible with the current level of specification [47].

2.8 Metalevel architecture in programming languages

A large amount of research concerning metalevel architecture comes from the programming languages domain. The most popular topic from this area is a reflection mechanism, whose understanding may differ between database and programming languages communities. In the former case, the main concern is the ability to extract database metadata in order to make it manipulable by generic applications, the latter focuses on behavioral reflection, including the ability to dynamically modify the program being currently executed. The major contribution of metalevel architectures is support for the postulate of *separation of concerns* in software development (credited to Dijkstra [12]). Two approaches to this issue are briefly presented below.

The Aspect Oriented Programming (AOP) [30] has recently gained a significant popularity as a programming paradigm improving the ability of traditional languages to deal with so-called *crosscutting concerns*. As the name suggests, those are the required features of software, which are difficult or impossible to modularize using classes, methods or procedures. They are often related with various kinds of non-functional requirements, like e.g. persistence, security, synchronization, real-time constraints or monitoring, whose implementation is usually scattered among different fragments of code, which makes it less readable and complicates its maintenance. The solution offered by the AOP is an abstraction called *aspect*, used to modularize a given concern in a way that does not affect the base code.

This provides a kind of additional design "dimension", which becomes integrated into the application during the compilation phase, by a mechanism called *aspect weaver*.

For example, the well known Java-based implementation of the AOP paradigm, called AspectJ [2], provides two general kinds of extending mechanisms:

- **Behavior modification using *advices***. The term *pointcut* denotes a declaration that identifies a family of precisely defined points in the execution of a program, which are called *join points*. AspectJ supports identification of *join points* of several kinds, including method call or execution, access to an attribute, and invocation of a constructor or exception handler. Those elements can be identified by their name (or its part), their location in terms of Java's classes and packages, as well as by signatures (that is, types of attributes or methods' parameters). *Join point* definition can be made more specific by combining several simple *pointcuts* using logical operators. *Advice* can provide arbitrary behavior to be executed before, after or even instead of execution points defined by a given *pointcut* it is assigned to.[10] Additional flexibility is provided by a reflective interface, allowing to extract a context of particular *join point* from within the *advice* code.

- **Class structure modification using *introduction*.** This mechanism allows for modifying definition of a given class, both in terms of behavior (new or overridden methods) as well as the structure (additional attributes). This includes possibility to modify inheritance hierarchy by making existing class extend other class or implement certain interface. All these changes can be achieved by declarations located outside the considered class.

Another approach to the separation of concerns problem is providing metaclasses as a mechanism fully definable by a programmer. In [14] the role of metaclasses is described through the analogy to natural language elements: if a class corresponds to a noun, then a metaclass can be compared to an adjective, as it provides certain properties (like e.g. *thread-safe* or *persistent*) further specifying a class definition. When a class is declared as an instance of certain metaclass, it gains its properties analogously like an object, whose behavior and internal structure is defined by its (regular) class. Similarly like in case of the AOP, it allows to decompose the problem according to different criteria (as every class can be an instance of multiple metaclasses) and to guarantee certain level of mutual independence if their redefinition. However, this approach may be lead to potentially very sophisticated structures of metaclass definitions, which may make the whole idea too complicated for a broader audience.

Since this work is focused rather on structural dimension of metadata, the majority of topics outlined in this section remain of little relevance to the subject.

[10] Of course, this may affect the consistency of original code. Note however, that the construct is much more controllable than e.g. the infamous **goto** statement, as the control is assumed to return to the original *join point*.

However, the issue of separation of concerns using metamodel constructs will be briefly discussed in further chapters.

2.9 XML Schema and the Resource Description Framework (RDF)

The World Wide Web Consortium (W3C) is a widely recognized organization involved in specifying common protocols for interoperability of web resources and services. The today's most popular specification of this consortium is the eXtensible Markup Language (XML), proposed to make web contents more meaningful, by introducing a logical structure through a use of HTML-like tags. Thanks to its extensibility and platform-independence, it also became an attractive option for porting a data between heterogeneous repositories. The XML is a base for a number of related technologies, concerned e.g. with visualization of contents, building links and relationships between XML entities, or defining the required structure of a XML document.

The last of mentioned issues is addressed by the XML Schema [63] specification. It introduces a concept of type, which may be a simple type or a complex type, where the latter describes a structure instead of a simple value. A type, once defined, may exist within a structure under a number of different fields. Field definitions may be described by multiplicities and default values. An unconstrained, predefined "Any" type is available. Based on existing type definitions, new, derived types can be defined, either through *restriction* of their values' domain or through *extension*, which means extending a complex type with a new element. Lists and unions based on simple types can be defined; for complex types union-like *choice groups* are available. Types can be defined abstract. A substitutability concerning derived types holds.

As a result of the XML design assumptions, the XML Schema does not deal with behavioral elements (like operations) nor with relationships (which are subject of separate XML-related specifications).

As can be seen from the above overview, the XML Schema is limited to imposing structural validity constraints on XML documents. Thus it leaves untouched the issue of providing semantic description of Web-based resources. This is the role of the Resource

Description Framework (RDF) specification, which is intended to provide a general mean to provide arbitrary descriptive information about web-based resources.

A resource, being a subject of RDF description is assumed to be uniquely identified by a Uniform Resource Identifier (URI), which may be constructed using URL. Resource descriptions are described through a graph structure, namely a directed, edge-labeled graph, where each described resource, called in this context *subject*, has a certain number of related named properties with their values. The part identifying such property is called *predicate* and has connected a value, which is called *object*. Since some *objects* may be *subjects* of further descriptions, the resulting graph may be arbitrarily complex. Since *objects* can be either simple values (e.g. strings), or related resources, the described mechanism effectively supports both resources' properties and relationships between resources [60].

Alternatively to a graph representation, a XML standard format for storing the RDF descriptions has been defined. It assumes storing each *subject-predicate-object* graph arc as a triple of XML statements, containing appropriate values and identifiers.[11]

Taking into account that the RDF is focused on resource's semantics instead of describing constraints on its structure and because the assumed resource-identification systems allows to deal with any describable entities (not just web documents), it can be treated as a lightweight ontology system to support the exchange of knowledge on the Web [21].

A prominent example of early application of the RDF is the Dublin Core Metadata Element Set (DC), specified by the Dublin Core Metadata Initiative organization. Although both specifications were developed independently, some requirements discovered when formulating the DC were taken into account in the RDF development [11].

The DC provides a very general collection of properties, which is intended to be applicable to a very broad range of application domains. It consists of the following 15 descriptive elements [11]:

- Title: A name given to the resource, by which the resource is formally known.

[11] To prevent ambiguity, also predicates can have their URIs assigned.

- Creator: E.g. a person, an organization, or a service, indicating the entity responsible.
- Subject and Keywords: The topic of the content of the resource. Using a controlled vocabulary or formal classification scheme is recommended.
- Publisher: An entity responsible for making the resource available.
- Contributor: An entity responsible for making contributions to the content of the resource.
- Date - associated with an event in the life cycle of the resource (it is usually the creation date).
- Type: The nature or genre of the content of the resource. Choosing a value from a controlled vocabulary is recommended.
- Format: Describes the physical or digital manifestation of the resource.
- Identifier: An unambiguous reference to the resource within a given context. Using a formal identification system like e.g. URI, DOI (Digital Object Identifier) or ISBN is recommended.
- Source: A Reference to a resource from which the present resource is derived.
- Language: A language of the intellectual content of the resource.
- Relation: A reference to a related resource.
- Coverage: The extent or scope of the content of the resource, in terms of place name, spatial location, temporal period of jurisdiction.
- Rights: Information about rights held in and over the resource.

As can be seen from the above example, the understanding of a term “metadata” differs significantly between the web content management and database communities. In the area of web resource description it tends to be a very broad term covering all related data not shown explicitly within a regular document. The database and UML community focuses on data element’s invariants and a description of the context that manipulates or accesses that data. The majority of DC’s properties would be applicable to particular data element (e.g. database object) rather than to a family of elements as it is typically the case for database metadata (e.g. *interface*, *type*, *collection*).

Nevertheless, a number of properties required for the content management are applicable to database metadata and thus may require metadata structure openness to accommodate new descriptive elements.

Although web resources tend to be less structured in comparison with traditional database management, RDF developers found it necessary to define, what descriptions (that is, properties) are expected for particular types of entities and perhaps also – what would be the types of their values. The resulting specification – the RDF Schema, is described as a vocabulary description language, defining classes and properties that can be used to describe other classes and properties [61].

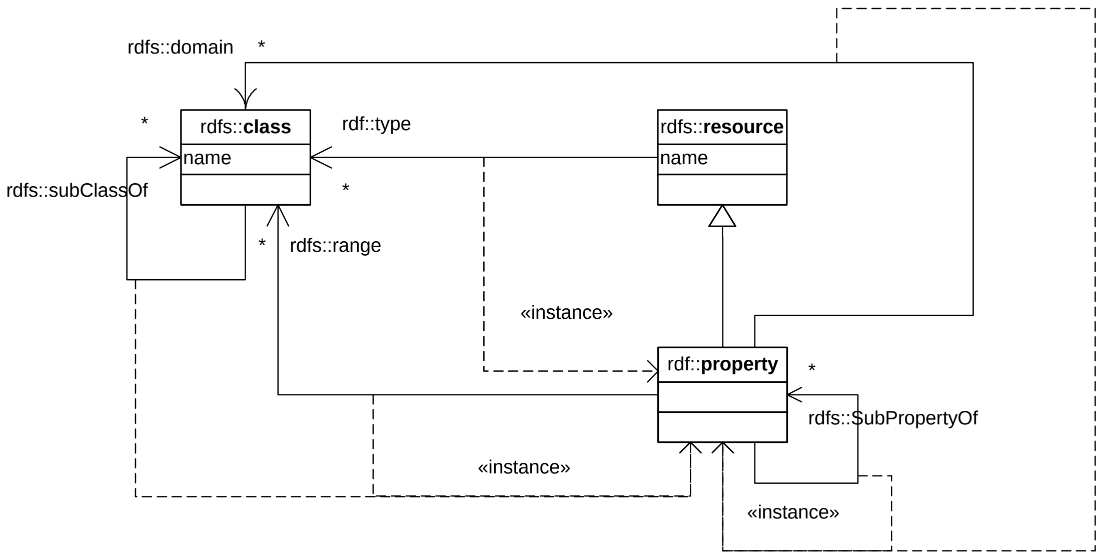


Fig. 4. RDF Schema concepts depicted using UML class diagram

The diagram in Fig. 4 is an attempt to show relationships among RDF Schema concepts. Since RDF descriptions constitute a kind of metadata, this directly corresponds to a metamodel concept discussed in previous sections. A *class* specifies a *type* of a given *resource* and determines which *properties* have been defined to describe it and what would be in turn the type of such description (through the *range* relationship). Moreover *classes* and *properties* can form generalization-specialization hierarchies, for which substitutability holds (as described in the context of XML Schema). The only element of this simple diagram that may be found disturbing is the existence of the *«instance»* dependencies suggesting, that the model mixes two metalevels (*range*, *type*, *subClassOf* and *subPropertyOf* are all instances of the *property*). In fact however, similar solution exists e.g. for the UML and the MOF, whose metamodels are defined in terms of their own core concepts, but it simply does not have to be explicitly shown in their metamodel diagrams as it is implied by the UML notation.

2.10 Objectivity/DB

The Objectivity/DB is a good example of a mature commercial ODBMS developed by an ODMG-member company. The system supports all three programming languages considered by the ODMG (C++, Smalltalk and Java) and its language bindings retain a relatively high level of compliance with the standard, although rather insufficient to guarantee a source code portability. Objectivity/DB does not support the OQL language at all, which is quite common among today's commercial ODBMSs. Instead, it provides only instance- filtering capability similar to the one specified by the JDO *Query* interface mentioned earlier.[12]

The following properties of Objectivity/DB are associated with its ODMG compliance:

- **Lack of stored procedures**. The drawbacks of this assumption have already been enumerated. The advantage is the ability to ultimately decentralize the processing (practically the only centralized feature remains transaction/locking control).
- **Usage of the data model of underlying object-oriented programming languages**. The absence of the OQL, makes a programming language the only mean to access database objects. The access is quite seamless, although at the same time rather low-level due to almost complete lack of declarative constructs.
- **Lack of object relativism**. Objects can only contain primitive members (not being objects) and object references. Therefore, although database can store structures of arbitrary complexity and nesting level, their implementational structure is quite flat. That is, the only real composition relation holds between objects and their attributes and among DBMS-defined storage objects. Except the composition hierarchy elements shown in Fig. 5, there is no other composite objects. This also entails inconveniences known from Java itself: the need of special treatment (also when

[12] It is even more limited than the JDO's solution due to the lack of support for query parameters and constraints over the attributes of type *Date*.

using reflection) and limitations connected with primitive types, make this "contamination" of object-oriented model rather unpopular [1].

On the other hand, in contrast to the ODMG definition, Objectivity/DB does not deal with the concept of *extent*. Instead, a persistence-capable object may be placed in any *container*, and each *container* may store instances of different classes. In effect, class definition is completely separated from storage structures. To better organize storage of very large amounts of data, the ODMG's concept of *Database* has been replaced by three levels of storage objects, as shown in Fig. 5. Moreover, some issues not addressed by the ODMG standard, like e.g. security mechanisms or event notification, has been designed for Objectivity/DB as services external to the core DBMS.

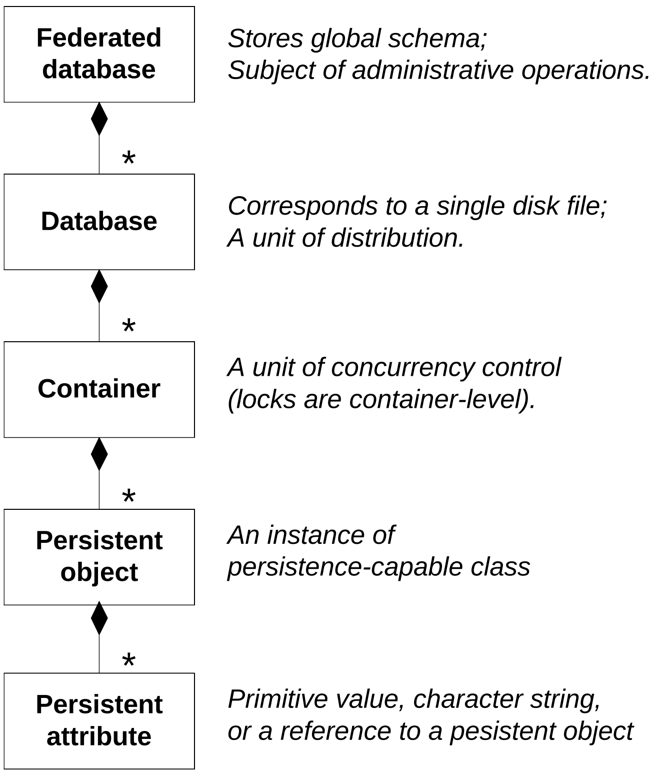

Federated database
Stores global schema; Subject of administrative operations.
*
Database
Corresponds to a single disk file; A unit of distribution.
*
Container
A unit of concurrency control (locks are container-level).
*
Persistent object
An instance of persistence-capable class
*
Persistent attribute
Primitive value, character string, or a reference to a pesistent object

Fig. 5. The containment hierarchy of the Objectivity/DB data structures

The system maintains database metadata internally. The ability to access or manipulate it differs significantly among programming language bindings. For C++ a special facility called *Active Schema* is offered as a separate product. Its interfaces, partly compliant with ODMG metamodel specification, allow for examining the schema and evolving it. It is also possible to browse and modify persistent objects through *Active Schema*, with absence of their class' implementation. The schema change history is maintained, but solely for the need of deferred object conversion in result of a schema evolution.

In case of the Java binding, the situation is completely different. The Java's reflective capability allows for isolating the schema from developer and for performing all its updates implicitly. That is, class is registered into a schema at the first time where the DBMS deals with persistent instances of such class or of classes whose definition refers to it (although classes can be also registered explicitly in advance). Despite the lack of Java interface to Objectivity/DB schema, some information can be extracted using standard Java's reflective API. This solution is limited though, since it requires presence of appropriate instances within database, as well as the access to the valid version of implementation of the classes registered in the schema. This is also a case for some of recently developed lightweight Java-based ODBMS (see eg. [10]).

Note also, that even in case of C++ interfaces, allowing access and updating of the schema definition, the schema structure is fixed and does not allow for any custom extensions of the underlying metamodel.

3 The concerns of DBMS metamodel

In this chapter the major issues related to the construction of the metamodel for object databases are presented. As will be shown, none of the currently available solutions (in particular, the ODMG standard), address all this issues in a satisfactory degree. A database metamodel should fulfill the following main roles:

- **Data Model Description.** The metamodel definition should be presented in a form that support understanding of the introduced data model by all parties, including system developers, users, administrators, researchers and students. It specifies interdependencies among concepts used to build the model, some constraints, and abstract syntax of data description statements; thus it supports the intended usage of the model.
- **Implementation of DBMS**. A metamodel determines the organization of a metabase (usually referred to as a system catalog in relational databases or a schema repository in object-oriented databases). It is internally implemented as a basis for database operations, including database administration and various purposes of internal optimization, data access and security.
- **Generic programming**. The metamodel together with appropriate access functions become a part of the programmer's interface to enable generic programming through reflection, similarly to Dynamic SQL or CORBA Dynamic Invocation Interface.
- **Schema evolution**. A metamodel equipped with data manipulation facilities on metadata supports schema evolution. As will be shown, this requirement is often not well understood. Changes to a schema imply a significant cost in changes to applications acting on the database. Thus schema evolution cannot be separated from software change and configuration management.
- **Separation of concerns at the metadata level**. Allowing the developer to modify the system behavior related with particular metadata, realized e.g. in the spirit of Aspect-Oriented Programming (AOP), seems to be very promising for the DBMS flexibility and its ability to deal with non-functional requirements. The metamodel definition should take into account the properties needed by such extensions.

- **Formal description of local resources** (called *ontology*) in distributed/federated databases or agent-based systems. A metamodel informs external parties (for example, mobile agents) on the content and organization of local resources. Recently this aspect has received special attention, reflected e.g. by the RDF standard of W3C [60] described in the previous chapter.

As will be demonstrated, these metamodel goals are contradictory to some extent. The following sections present peculiarities and requirements connected with each of the aforementioned goals. In the context of these issues the serious drawbacks of the ODMG standard metamodel proposal are identified. This will lead to general conclusion that the ODMG metamodel specification needs significant improvements.

3.1 Data model description

This role of metamodel is central in case of modeling language. However, also for an ODBMS metamodel it is of primary importance, especially because the clarity and understandability of new technology is critical for the success of its adaptation. This role of metamodel definition is also the most straightforward and intuitive. The usability of other metamodel features is dependent on the quality of this definition. Thus the clear description of data model primitives and their interrelations is necessary.

The UML [41] is an example of addressing that aspect of metamodel. This case shows that even without resorting to formal definitions it is possible to satisfactorily describe the meaning and the intended usage of data model notions. Graphical notation, together with a number of examples and natural language descriptions keep the number of ambiguities low. Taking into account that in case of database system metamodel constructs are connected with implementational structures and clarified by the query language semantics, the similar style of description would be satisfactory for DBMS metamodel. However, the fact that such metamodel need to be directly implemented speaks in favor of employing a much simpler metadata structure than the one implied by the UML.

It is necessary to mention that the quality of the ODMG metamodel description is much worse than in case of the UML. There are flaws in the style that the ODMG use to explain the goals and semantics of the metamodel, together with a lack of many definitions and explanations. Methods to access and update a metabase are not described at all. Thus, ODBMS developers must induce the meaning from names and parameters used in the specification, which will probably lead to incompatible (or non-interoperable) solutions. In its present form, this part of the standard is underspecified and ambiguous, thus making it difficult (or impossible) to understand the intended usage of its features.

3.2 DBMS schema implementation

This section introduces the requirements that are to some extent contradictory to those connected with descriptive role of metamodel. Definition of a metamodel following the UML style guarantees expressiveness, however such a rich structure is impractical or even unacceptable concerning the implementational requirements. The database schema implementation brings the following criteria of metamodel quality:

- **Simplicity**. The metamodel and the metabase should be simple, natural, minimal and easy to understand, in order to be efficiently used by developers of DBMS and database administrators.
- **Universality**. Implementation of database languages and operations requires various accessing and updating operations to the metabase. The metamodel should support all such operations, and these operations should match, as closely as possible, similar operations for regular data.
- **Performance**. Metabase operations that originate from the database management system or from applications may be frequent, and thus it is important to organize the metabase so as to guarantee fast run-time access and updating.
- **Physical data structure information support**. Data describing physical structures (e.g. file organizations, sizes of collections, indices, access methods, etc.) as well as data used for optimization (access statistics, selectivity ratios, materialized views, stored results of methods, etc.) must be included in the metabase. Although this information is not relevant to the database conceptual model, the metabase is the only place to store it. Thus, it would be appropriate to allow extensions to the metamodel structure, to provide storage for all necessary information regarding the physical properties of a database.

- **Privacy and security**. As stated previously, this aspect is not relevant to the database conceptual model, but the metabase repository is usually the place to store information on privacy and security rules. The metabase repository itself should also be a subject for strong security rules.
- **Extensibility**. The metabase structure and interfaces should be easily extendable to support further development and extensions of DBMS functionalities. There are features such as views, constraints, active rules, stored procedures, etc. which could be incorporated into future ODBMS standards and implementations.

In this role the metamodel presented in the ODMG standard is too complex: 31 interfaces, 22 bi-directional associations, 29 inheritance relationships and 64 operations. It is too difficult to understand and use by programmers. The worse, this already large structure is by no means complete. There are many examples showing that the defined methods are not able to fulfill all necessary requests. Some of abovementioned features requiring standardization (e.g. privacy and security) have not been addressed so far. Some other (e.g. some kinds of physical data structure information) may need to be a subject of database vendor's custom extensions that would further complicate the metamodel structure. Moreover, future extensions of the standard, such as rules and triggers, views, database procedures, methods (not covered by the ODMG standard) will cause further growth in the complexity of the metamodel. Summing up, the ODMG metamodel structure is very complex and at the same time far from being complete. This leads to the conclusion that this style of metamodel definition would result in an interface too complicated to be effectively used by programmers.

3.3 Generic programming through reflection

As explicitly stated, the ODMG metamodel should have the same role as the Interface Repository of the CORBA standard, which presents some data structures together with operations (collected in interfaces) to interrogate and manipulate the defined IDL interfaces. The primary goal of the Interface Repository of CORBA is dynamic invocations, i.e. generic programming through reflection. This goal is not supported by ODMG, because the standard does not define all necessary features [47].

Nevertheless, such reflective capabilities proved to be useful e.g. in SQL and should certainly be considered in a standard for ODBMS. Generic programming through reflection requires the following steps:

1. Accessing the metamodel repository to retrieve all data necessary to formulate a dynamic request.
2. Construction of the dynamic request, e.g. as a string, representing a (parameterized) query.
3. Executing the request (with parameters). This assumes the invocation of a special utility, which takes the request as an argument. The result is placed in a data structure specifically prepared for this task. Since a request is usually executed several times, it is desirable to provide a preparation function that stores the optimized request in a convenient run-time format.
4. Utilizing the result. In more complicated cases the type of result is unknown in advance and has to be determined during run time by a special utility that parses the request against the metamodel information.

Although the ODMG standard specifies access to meta-information, thus supporting step 1, it does not provide any support for the subsequent steps (for a detailed discussion see [47]).

The four reflection steps are implemented in dynamic SQL (SQL-89 and SQL-92) and in CORBA DII. Of special interest are the requirements for step 4. For the result returned, it is necessary to construct data structures whose types have to be determined during run-time. A query result type can constitute a complex structure, perhaps different from all types already represented in the schema repository. This structure can refer to types stored in the schema repository. Moreover, it must be inter-mixed (or linked) with sub-values of the request result, because for each atomic or complex sub-element of the result, the programmer must be able to retrieve its type during run time. Hence the metamodel has to guarantee that every separable data item stored in database is connected to information on its type and this information must be available after query execution. Construction and utilization of such information presents an essential research problem.

Similarly, access to a metamodel repository is necessary to determine the structure of parameters required by the constructed request. Such features are available in SQL through the “describe” statement. In contrast to the relational model, an object model has to deal with arbitrarily complex types of query results. The ODMG standard does not specify this aspect of the metamodel, thus a substantial subset of generic programming tasks are not implementable.

To allow generic programming and portability, the standard (including its metamodel definition) must be very precise in each of the abovementioned aspects, including standardization of programming facilities concerning steps 2, 3 and 4. Even subtle differences in the organization of database repositories, their access operations or request execution functionality, undermine the portability of generic applications.

3.4 Additional schema elements and extensibility

As already stated, a database schema has to store also a number of items not directly reflected in the data model. In particular, additional information is needed to support data storage. Additional elements may concern information on physical database structure. Those of them (e.g. the number of elements in collections) that could be explicitly accessed by application developers, have to be defined in the standard. Some others, e.g. presence of indices, different kinds of data access statistics, etc., could be the subject of extensions proprietary to a particular ODBMS.

Another example of additional metadata elements are information on ownership and access permissions. Since such mechanisms are built into the DBMS and accessed by applications, appropriate metadata elements should be the subject of standardization (c.f. CORBA Security Service [40]).

In contrast to the relational model, type definitions in object systems are separated from data structures. Hence a metamodel repository must store definitions of types/classes/interfaces as distinguishable features connected to meta-information on storage structures.

3.5 Schema evolution and Software Configuration Management

Schema evolution capability is an important feature of a modern DBMS and is one of the most prominent issues to consider during database metamodel design. In contrast to the majority of the papers devoted to this subject, which are focused on validity of schema modifications and on the subsequent object conversions, this work emphasizes the Software Configuration Management (SCM) aspect of a schema change. Thus, in this section the different issues of software configuration management are summarized and their relevance to DBMS itself is investigated.

Schema evolution in object databases

As a prominent feature of modern DBMS, schema evolution is supported in a number of commercial products. Unfortunately, this important aspect of ODBMS functionality is not effectively standardized and thus it is realized through proprietary solutions. The ODMG standard touches this issue only implicitly and, a will be shown, for different reasons inadequately. It may be surprising that the standard does not deal with this issue, especially, taking into account that the interfaces used to define its metamodel provide the modification operations. Their presence is adequate only in the context of schema evolution.

This aspect of database functionality has been present for a long time as one of the main features to be introduced in object-oriented DBMS [4] and its importance is unquestionable. Although the database literature contains over a hundred papers devoted to the problem (e.g. [8],[18],[43],[45],[58]), it seem to be far from being solved. The majority of these proposals, although inspiring, can be perceived as too idealistic for today's software development practice. Taking a more pragmatic approach, this section presents the problem from the software engineering point of view.

Obviously, the schema evolution problem is not reduced to some combination of simple and sophisticated operations on the schema alone. After changes to a database schema, the corresponding database objects must be reorganized to satisfy the typing constraints induced by the new schema. Moreover, application programs acting on the database must be altered. Only some simple changes, such as the addition of a new class, association, attribute, procedure or method do not impact on existing applications, and only in the case of a proper level of data independence. Naive approaches reduce the problem to operations on the metadata repository. This is a minor problem, which can be solved simply (with no research), by removing the old schema and inserting a new schema from scratch. If database application software is designed according to software configuration management principles, then the documentation concerning an old and a new schema must be stored in the SCM repository. Hence, storing historical information on previous database schemata in a metadata repository (as postulated by some papers) in the majority of cases is useless.[13] In fact, apart from few very specific applications it is necessary to maintain at each moment exactly one valid schema version, with the requirement of consistent handling the changes applied to the schema. On the other hand, serious treatment of SCM and software change management excludes ad hoc, undocumented changes in the database schema.

The ODMG solution (as well as many papers devoted to schema evolution) neglects the software configuration and software change management aspects. This seems to result from the fact that although the DBMS construction and software configuration management constitute the well established areas of research, they are usually considered in separation from each other. To effectively support the schema change in larger systems, a DBMS should provide features for storing dependency information concerning the schema. The advantage of such solution over storing them together with other configuration data within an SCM repository, would be the ability of automatically discover and register dependencies concerning database elements. This would require new metamodel constructs dedicated to this role.

Assuring system's consistency after schema change

Maintaining the consistency between regular data and metadata is the most obvious requirement in the context of schema evolution. This is also relatively easy to realize, as both elements are managed solely by a DBMS. Therefore, the most significant practical problem related to schema management is the schema change impact on database-dependent applications. This has been recognized and different attempts to eliminate that issue were made.

Many papers assume that the problem can be solved by database views. After changing a schema one can define views, which provide virtual mappings from the existing objects to the new schema; hence no changes occur in database object and no changes in existing applications is required. Alternatively, one can convert existing objects according to the new schema, but together defines views, which preserve the old schema for already defined applications. In both cases, old applications need not be altered, hence the major problem of schema evolution is solved.

[13] Nevertheless, some parts of schema changes history may need to be maintained by DBMS internally, to perform the deferred object conversion consistently (see e.g. [33]).

In the majority of cases such an approach is idealistic for the following reasons:

- Some changes in a schema stem from unsatisfactory properties of applications, hence changes of applications are inevitable.
- Some changes in a schema stem from changes in business data ontology (e.g. implied by new law regulations). Any automatic mapping of existing data is unable to fulfill new business requirements.
- View definition languages are not sufficiently powerful to cover all possible mappings. There are many mappings not covered by SQL views.
- The view updating problem is solved only in specific cases and (probably) will never be solved in the general case. Hence many applications that require advanced view updating, cannot rely on this approach.
- Access to data through views may result in unacceptable degradation in performance. Although materialized views partly solve the problem, this approach implies disadvantages: an additional storage, the updating of materialized views after updating of the original database, the updating of the database after updating the view.
- Applications written in languages such as C++ and Java are tightly coupled to physical properties of database objects. Many (sometimes undocumented and low-level) dependencies between application programs and database object limit the use of database views.

In summary, although database views provide some hope for schema evolution, this approach is non-applicable in majority of cases. More detailed discussion on this topic can be found in [56].

Another approach to schema evolution can be based on concepts such as wrappers and mediators. The conceptual border between wrappers and mediators is undefined - it is usually assumed that wrappers implement simple mappings and mediators possess some “intelligence”. The approach is similar to the approach employing database views, but in contrast to database views, which are defined in high-level query languages (SQL), wrappers and mediators are proprietary solutions, tightly coupled to a category of applications and written in lower-level languages (C/C++, Java, etc.). The approach is more realistic than the approach based on database views, but it requires low-level programming. It is extensively used in CORBA-based environments (in CORBA terms, wrappers and mediators are covered by the concepts of adapter and skeleton). Rules for designing and writing wrappers and mediators are not formalized or disciplined: each problem requires a standalone solution. Some of the abovementioned disadvantages of using views are also true for the approach based on wrappers and mediators. In particular, if a change concerns data representation or business data ontology then any kind of wrapper or mediator may be unable to solve the problem. In any case, the change will affect applications.

Concluding, there is probably no satisfactory solution that would effectively isolate application programmers from the change impact. Thus handling this aspect of software change by resorting to the SCM methods would be inexplicable.

Schema evolution in software change management

Schema evolution forms part of a more general topic, which is referred to as *software change management*. It concerns the maintenance phase in the software life cycle. The cost of maintenance is very high and in total can, by several times, exceed the cost of initial software development. Thus, some discipline is necessary to reduce the cost. Software change management provides activities during the software development, operation and maintenance to support software changes. It also determines disciplined processes for altering software after changes. Both of these aspects are important. If software developers and clients neglect certain activities addressing future (usually unknown) software changes, then the future cost of changes can be extremely high. Changes to the software should follow some life cycle to reduce cost and time, and to achieve proper software quality.

Basic activities of the software change management during software development and operation are the following (among many others):

- Keeping high quality, availability and up-to-dateness of user and system requirements, as well as analytical and technical documentation.
- Keeping clarity of software conceptual models, including the database schema and the structure of software modules.

- Preserving proper software architecture, which reduces contamination of software changes (for example, three-tier or multi-tier architectures).
- Precise specification of interfaces between software modules.
- Supporting software reuse by the identification and specification of reusable assets and generic modules (e.g. templates); supporting reuse by object-oriented methods of software analysis, design and construction.
- Using high quality and high-level abstraction of software tools, avoiding programming in low-level languages, avoiding proprietary solutions and hybrid, eclectic architectures.
- Preserving software quality by following quality assurance procedures and standards.
- Software configuration management, which includes safe storage of all documents which appear during software development (source codes, requirements, technical documentation, etc.), preserving completeness and availability of the documents, keeping information on software versions, and keeping completeness and mutual consistency of all documents within a version (including historical versions).

The software change management must also provide activities to accomplish organized software change processes, in particular, the following:

- Organizing the process of reporting problems in software and/or in (changing) user or system requirements.
- Collecting and storing software problem reports; organizing processes in which some organization units (e.g. a software change committee) are responsible for assessing reports and qualifying them according to importance, urgency, potential cost and impact on other software modules.
- Organizing preliminary diagnosis of software problems and cost estimations of software changes.
- Decision processes concerning the scope of software changes and/or making new versions of the software.

- Planning software changes, including feasibility studies, estimating costs and time, scheduling, organization of software development team responsible for the changes, organizing software quality assurance processes, testing of the changed software, regression testing of unchanged software (which can potentially be influenced by the change), documentation, SCM, etc.
- Organizing the design and implementation of software changes.
- Organizing testing changed software according to software testing plan.
- Organizing regression testing (testing unchanged modules that can be influenced by the change).
- Documenting changes, including software requirements, analytical, technical and user documentation.
- Installation of changed software, training of users and acceptance tests.
- Learning from change, and collecting historical data (cost, time, etc.) concerning software changes, to improve the change processes in the future.

A proposal concerning schema evolution should refer to the activities of software development presented above, to determine a clear goal for the research. It can be formulated in terms of cost, time or quality of particular activities, and/or in terms of software quality.

Software Configuration Management

Schema evolution is closely related to another area of software engineering, namely the Software Configuration Management (SCM). SCM is a discipline for establishing and maintaining the integrity of the products of a software project throughout the project's lifecycle [22],[23],[24],[25],[26]. SCM consists of planning, organizing, surveillance, controlling and coordinating activities, making it possible to identify, store and secure all components of the software and its documentation during the entire software life cycle, including change management. SCM is especially important if a project lasts several years and/or has many versions due to changing user requirements or system requirements. Bad SCM can totally paralyze a project. Schema evolution means a new version of a schema and, in consequence, a new version of the database, and a new version of applications. Thus, it must be disciplined by SCM.

A basic entity that SCM deals with is a software configuration item (SCI). An SCI can be atomic or complex. Complex SCIs include all software artifacts that present intermediate or final software products, including source code, documentation, tools, tests, etc. The basic principle behind SCM is that SCIs must be consistent. SCIs frozen for changes are called baselines. Some SCIs are called versions, revisions and releases.

All software entities that are used or produced within a particular software version must be collected together as SCIs and stored carefully. This makes it possible to avoid situations where new code is associated with old documentation; old code cannot be re-compiled because a relevant older compiler version is no longer available, etc.

The scope of SCM concerns:

- Documentation: user requirements, system requirements, analysis and design documents, testing documents, software quality assurance documents, etc.
- Modules with source codes, object codes, program libraries, and binary codes.
- Designed graphics, user screens, Web pages, etc.
- Text files, dictionaries, databases.
- Workspace tools: hardware, compilers, linkers, interpreters, protocols, libraries, RAD tools, CASE tools.
- Software target hardware and software configurations (as documents or scripts).
- Testing configurations, data, software and results.
- Change management documents: new user requirements, new system requirements, software problem reports, decisions of the software change committee, new code, results of testing, etc.

As follows from the above, all versions of a schema must be the subject of SCM. The new schema must be stored within a consistent configuration which includes new requirements, diagnosis and analytical documentation, data conversion code, code of new application modules, new design and implementation documentation, testing code, data and results, software transfer documentation, user documentation, etc. Schema evolution cannot be separated from other SCM aspects and activities.

The fact that schema evolution is a part of SCM has consequences for the metamodel. Usually, SCM implies the existence of a special repository for organizing and storing all software and documentation entities that are the subject of SCM. In simplest forms SCM is accomplished through packages such as CVS or Rational's ClearCase, used by project teams to keep track of code and documentation versions. In more advanced cases, SCM is based on specially designed software libraries with complex structures storing information on software modules, documentation, configuration items, versions, projects, persons, roles in projects (project managers, analysts, designers, programmers, etc.), physical locations (file directories, databases, rooms, shelves, CD ROMs), etc. Such a library is related to another important activity of a software company known as knowledge management (with emphasis on tacit knowledge of people - participants of projects).

It may be assumed that all physical software entities/documents are stored as *library items* (including source codes, documentation, compilers, DBMSs, operating systems, CASE tools, etc.). *Configuration items* are logical structures built over references to library items, to bear information on *configurations*, i.e. consistent sets of software entities and documents. Some configuration items are *baselines*, i.e. frozen to changes. Some baselines are *releases*, i.e. ready software products that are installed for clients. Physical documents can be of several kinds: software components, management documents, software quality assurance documents, working documents, etc. Configuration items and library items can be the subject of activities: creating, deleting, changing, accepting, inserting new items, etc. Activities are performed by project roles or persons. A project role is responsible for some configuration items. A person has access rights and can loan a paper document or lock an electronic document (to prevent simultaneous changes). All historical activities, loans and locks are kept in the repository, to enable restoring the history of changes.

It is implicitly assumed in the research devoted to schema evolution (in particular, in the ODMG standard) that actions on the database schema repository will immediately change a repository state. Sometimes, it is assumed that the repository will also be prepared to keep historical information on previous schemata. Taking into account software change management and SCM, such an approach is inadequate. A change to a database schema must be carried out on the SCM repository, which should be prepared to keep both old and new schemata. Many other documents are related to this change, including software problem reports, new requirements, managerial decision documents, diagnostic documents, analytical documents, design documents, software code, testing documents, etc. All of this information must be kept within an SCM repository rather than within a metabase repository. The following subsection identifies other tasks assignable to a metabase that would provide an important support of the SCM while avoiding redundancy resulted from overlapping responsibilities.

Dependencies among software units

Some tasks of SCM are more efficient (in terms of cost, time and quality) if the information on dependencies between software units could be properly organized. This concerns the following tasks:

- **Diagnosis of a problem**: this information makes it easier to conclude which part of the software is responsible for the problem or which part of the software is coupled with an old requirement that needs to be changed.
- **Planning and scheduling**: the dependencies can show the scope of changes hence make it possible to estimate the cost, time, staff, infrastructure, etc. necessary to introduce a change.
- **Implementing and testing**: implementing a change requires (as a rule) knowledge of dependencies of a changed software unit with other units.
- **Regression testing**: after a given unit is changed and tested it may happen that some other units are affected. Regression testing means repeating testing processes on unchanged units in order to confirm their validity.
- **Preparing new documentation**: after a change, further changes must frequently be introduced to the existing software documents;
- **User acceptance tests and education**: after a change it is necessary to recognize system functionalities which were affected by the change, thus requiring new user acceptance tests and education;
- **Configuration updating**: after a change it is necessary to establish a new baseline or a version (revision, release), i.e. consistent SCIs to be approved by official

bodies. This requires recognition of all software components and documents that had been affected by the change.

Some dependencies between software units are or can be stored within a metabase repository. Other dependencies can be stored within an SCM repository, in particular, as SCIs. Some dependencies are difficult to recognize, thus they may require some extension of a metabase repository or an SCM repository. Below more important dependencies are listed.

- **Configuration dependency**: some software and documentation units are dependent because they create a consistent SCI. This dependency is usually stored within a configuration repository. Configuration dependency is more relevant to SCM.
- **Standardization dependency**: which software items follow the same standards. The dependency can be considered a particular case of the configuration dependency if standards are stored as configuration items. Standardization dependency is more relevant to SCM.
- **Forward dependency** between procedural units of the software (procedures, functions, methods, views, etc.). The dependency shows which procedural units are called from a given procedural unit. This dependency is easy to discover by analysis of the code. The dependency can be stored within a metabase as e.g. a kind of UML collaboration diagrams showing message flow between classes/interfaces. Forward dependency is relevant to a metabase.
- **Backward dependency** between procedural units of the software. The dependency is exactly reverse to the forward dependency. It is more valuable than the previous one because it shows which software units call a given unit. As stated previously, dependency can be stored within a metabase and associated with proper utilities. Backward dependency is relevant to a metabase.
- **Parametric dependency**: it shows dependency between a given unit and a unit that can be a parameter to the given unit. This concerns e.g. *call-by-reference* parameters of methods or parameters of some (generic) software templates. Parametric dependency is relevant to a metabase.
- **Side effects dependency**: Side effects concern all aspects of the data/computer environment that can be affected by a given procedural software unit. Side effects

concern operations on a database, global data (shared among applications), environment variables, hardware devices, operating system registers, catalogs, files, external communication (ports, Internet), etc. In languages such as Modula-2 and DBPL some side effects are explicitly determined by special programming facilities called import lists. Current object-oriented languages do not determine side effects within class interfaces, hence the programmer and the system is unable to recognize them directly. This can be the source of serious bugs, cf. the Ariane-5 rocket disaster caused by an unspecified side effect. Side effects can be passive (a given procedural unit reads the state of some external resources), or active (a given procedural unit affects the state of some external resources). A metabase repository can store information on side effects; providing the information on them is an obligatory part of a software unit specification. For instance, a given method can be associated with a part of database that can be read and updated. Side effects dependency is relevant to a metabase and SCM.

- **Event dependency**: holds between a unit raising an event and a unit catching it and triggering some action. The case is similar to forward and backward dependency. This information is usually present in the specification of interfaces (CORBA IDL, ODMG ODL), thus it can be stored within a metabase repository. Event dependency is relevant to a metabase.
- **Definitional dependency**: holds between two data units, where one is a definition and another one is an instance of this definition. The dependency concerns interfaces, classes, types, patterns, skeletons, templates, schemas, specifications, etc. and their instances. Definitional dependency is relevant to a metabase and SCM.
- **Redundancy dependency**: holds between units that contain redundant information; for example, dependency between copies of some data that are introduced to improve performance or to increase safety. Redundancy dependency is relevant to a metabase and SCM.
- **Stylistic dependency**: holds between software units that follow the same design style of a user interface; for example, screens shown to the user, manipulation paradigms, used terminology and metaphors, etc. Stylistic dependency is more relevant to SCM.

Taking into account the entire population of software and documentation units, their structure can be expressed as a (partly directed) colored graph, where each edge represents some dependency between units and the color of an edge represents a dependency kind. Some dependencies in this graph form subsets of software/documentation units, in particular, configuration dependency, definitional dependency and stylistic dependency. Some dependencies can be stored within a metabase repository. Other dependencies are more relevant to a software configuration repository.

In summary, the properly defined schema evolution problem should establish dependencies between software and documentation units. It should clearly subdivide the dependencies between a metabase repository and a configuration repository, and should clearly determine benefits of storing the information on dependencies for particular phases and aspects of the software life cycle, including the software change management. None of the abovementioned aspects of schema evolution are taken into account in the metamodel defined by the ODMG standard. This suggests that the approach to schema evolution assumed by ODMG does not follow the principles of professional software development. Unfortunately, the last statement also concerns the majority of papers devoted to schema evolution.

Concluding, the schema evolution problem far exceeds the pure problem of metadata management and should be considered as a part of software change and configuration management. While some repository-updating operations would indeed be useful, e.g. adding a new attribute or adding a new class, the operations do not solve the essential problem. The major problem of schema evolution concerns altering a database and – most of all – altering applications that operate on the database. This problem is related to software engineering rather than to the pure database domain.

3.6 Separation of concerns

The importance of this issue (outlined in the previous chapter) seems to be unquestionable. However, in case of DBMSs two factors need to be noted. Firstly, a DBMS is by its nature a tool less generic than a general-purpose programming language. In effect, some of the features being potentially a subject of aspects specified by a programming language are fixed in a DBMS in a form of its internal mechanisms.

Secondly, the main issue of an ODBMS metamodel is its overall complexity. An aspect-supporting mechanism may appear not worth of the cost of maintaining additional constructs. Those pros and cons are further investigated in the next chapter.

3.7 Ontology

As already suggested, it is also necessary for a database schema to include information forming its ontology. Even if such information were to be used during human-assisted resource discovery and integration rather than by autonomous agent software, it is obvious that much more information than just structural constraints and interface signatures is needed. However, such descriptions seem to be to a large extent domain-specific and thus difficult to fix in the form of standardized format. Thus the description structure should be open and allow for unambiguous vocabulary specification, as suggested by the RDF specification outlined in the previous chapter [60]. Having specified a flexible structure of such description, a choice concerning vocabulary standardization can be made. The options are either a very general property set like the Dublin Core Metadata Element Set [11] or a family of domain-specific profiles in the spirit of the OMG Domain Specifications, or a combination of both approaches.

4 Proposed features of the DBMS metamodel

In this chapter, some general directions for the construction of database metamodel are presented. The aim is to make it as simple as possible, flexible and open for future extensions. Some of the more specific features and issues of such metamodel are also addressed.

4.1 *Metadata manipulation language*

There is an opinion that the SQL language has been the key feature that brought the broad acceptance of relational databases. At the same time the limitations of the ODMG's OQL (Object Query Language), a very limited support of this language by commercially available ODBMS and definitely secondary role of that language assumed by the vendors, seem to be an important obstacle for popularization of the object databases [7].

For this reason, the general assumption of this work is that a fully-fledged query language equipped with imperative constructs is necessary as the main (and usually sufficient) mean of implementing database applications [54]. This would make the general-purpose programming languages an auxiliary tool, needed only to realize a specific tasks not supported by the database language. Such a big conceptual change makes many of the solutions suggested in this work further from the current ODMG proposal than they could be. Moreover it raises questions about the feasibility of a wider acceptance of a new language. Anyway, such change seems to be indispensable in order to rebuild the currently low-level object database interfaces, to make them comparable to powerful tools available for relational database systems in terms of programmer's productivity.

The usage of such language would of course also concern schema. A standard generic set of operations for metadata search and manipulation (together with their allowed usage scenarios) should be defined. A predefined set of methods is a bad solution as it contributes to the metamodel complexity while not guaranteeing the completeness of functionality. Such an approach is assumed in [51], where a special metamodel language MetaOQL is proposed. However, after defining catalogs as object-oriented structures, they can be interrogated by a regular OQL-like query language, extended by manipulation capabilities, e.g. as proposed in [55]. Because the structure of the catalogs can be recursive, it is essential to provide corresponding operators in a query/manipulation language, such as transitive closures and/or recursive procedures and views.[14] These operators are not considered for OQL. So far, only the object query language SBQL of the prototype system Loqis [57] fully implements them. Such operators are provided for SQL3/SQL1999 and some variant of them is implemented in the Oracle ORDBMS. Moreover, to make database applications portable the high-level catalog structure must be the subject of the standard.

The above suggestions support the assumed simplicity and minimality of programmer's interface. A similar solution is provided by the SQL-92 standard for relational databases, where catalogs are organized as regular tables accessed via regular SQL. Using the same constructs to access the database and the metamodel repository would not only make it easier for programmers, but would also be advantageous for performance due to utilizing a query optimizer implemented in the corresponding query language.

In case of data items that are to be accessed in a number of ways it is critical to provide a fully universal generic interface. Even an extension to the current collection of methods proposed by ODMG cannot guarantee that all requests are available. Moreover, programmers should be able to create their own access/manipulation abstractions (methods, views, procedures, etc.) and store them as a part of the metadata repository. In summary, this suggests the solution, where the metadata repository could be interrogated and processed by a universal query/programming language *a la* PL/SQL of Oracle or SQL3/SQL1999. This solution also addresses a part of the schema evolution topic that deals with changing the state of the metabase repository.

4.2 Simplifying the metamodel

As it was already emphasized, the main problems with the current metamodel definition from ODMG results from its size and redundancy, making it too complicated for implementation and usage by programmers. There are two general means to reduce the complexity of metamodel access. Firstly, the removal of some inessential concepts

[14] Again, such features are applicable also to regular data. This leads to the conclusion that the metadata and regular data can be accessed in a uniform way, except for additional constraints on manipulation of the former.

can be considered. Secondly, the remaining concepts should be arranged into a structure guaranteeing simplicity of its access and flexibility of its future modifications.

Minimality of a metamodel

There are several options to reduce the overall number of metamodel concepts. The simplest improvement in this direction is the removal of concepts that are redundant or of limited use. For instance, the removal of the *set* concept can be considered, because the *multi-set* (*bag*) covers it and applications of sets are marginal (SQL does not deal with sets but with bags). Another recommendation which can considerably simplify the metamodel (as well as a query language) concerns object relativism. It assumes uniform treatment of data elements independently of a data hierarchy level. Thus, differentiating between the concepts of object, attribute, subattribute, becomes secondary. Some simplifications can also be expected from the clean definitions of the concepts of *interface*, *type* and *class* and their interrelations.

An important source of redundancy of the ODMG standard is the approach aimed to directly support different language bindings. This is another argument in favor of introducing a single, unified database language in the spirit of PL/SQL as a main and self-dependent mean of manipulating both regular data and metadata.

Flattening a metamodel structure

The basic step toward simplifying the metamodel definition concerns flattening its structure. Separate metamodel constructs like *Parameter*, *Interface* or *Attribute* can be replaced with one construct, say *Metaobject*, equipped with additional meta-attribute *kind*, whose values can be strings “parameter”, “interface”, “attribute”, or others, possibly defined in the future; Fig. 6.

This approach radically reduces the number of concepts that the metadata repository must deal with. Moreover, it supports extensibility, because a new concept means only a new value of the attribute “kind”. The metabase could be limited to only a few constructs, as demonstrated in Fig. 7. Although this meta-schema does not support some useful concepts (e.g. complex and repeating meta-attributes, attributes of meta-relationships), it constitutes a sufficient base for the definition of the majority of constructs provided by the ODMG metamodel. To provide complex/repeating meta-attributes the meta-values can be extended in the XML style.

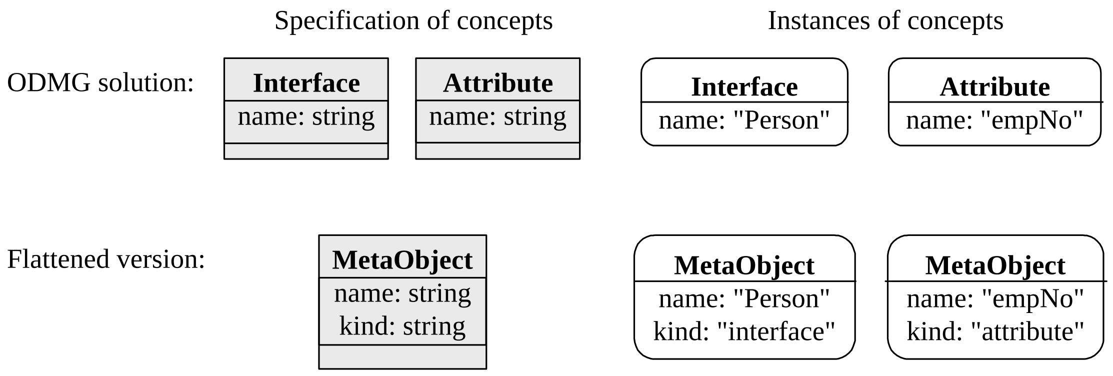


Fig. 6. Original and flattened ODMG concepts

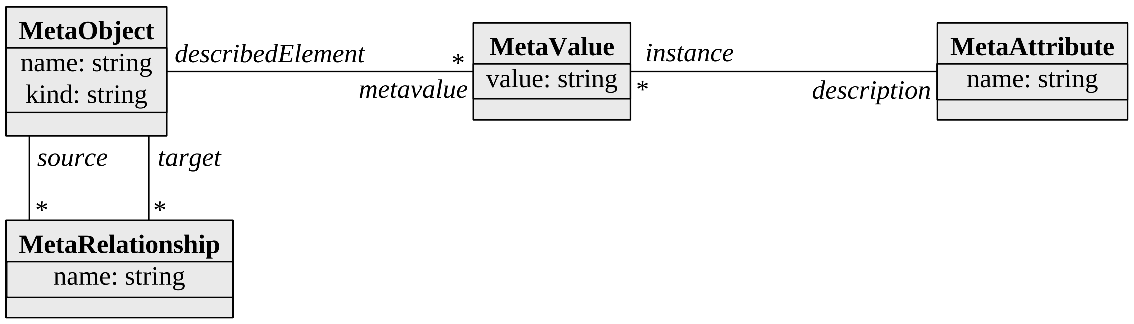


Fig. 7. Concepts of the flattened metamodel

Fig. 8 presents a simple ODL schema and Fig. 9 and Fig. 10[15] present one possible state of the schema repository according to the metamodel presented in Fig. 7.

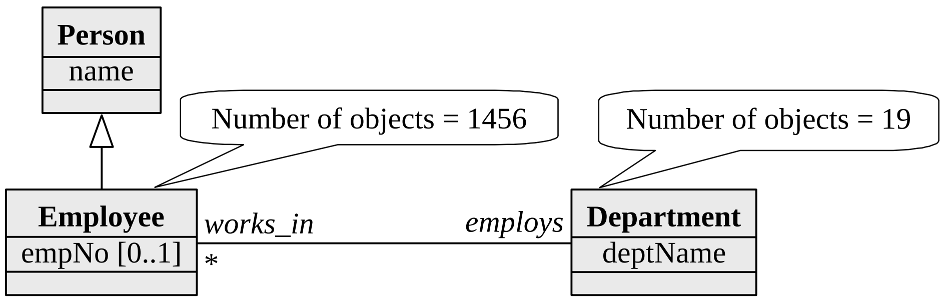


Fig. 8. A simple ODL schema

The large part of the presented metadata is used to define appropriate object data model constructs. In order to define a standard metamodel, the flattened metamodel has to be accompanied with additional specifications, which should include:

[15] Those examples use metadata concepts following the original ODMG terminology.

- Predefined values of the meta-attribute “kind” in the metaclass “MetaObject” (e.g. “class”, “interface”, “attribute”, etc.); they should be collected in an extensible dictionary.
- Predefined values of meta-attributes “name” in metaclasses “MetaAttribute” (e.g. “count”) and “MetaRelationship” (e.g. “specialization”).
- Constraints defining the allowed combination and context of these predefined elements.

A standard metamodel should define the aforementioned values and constraints of an object data model together with the most important additional data elements required by functionalities of an ODBMS schema.

Flattening the metamodel makes it possible to introduce more generic operations on metadata thus simplifying its usage by designers and programmers. Flattening also supports extendibility, as it is easier to augment dictionaries than to modify the structure of meta-interfaces. Simplification of the metadata structure can support the run-time performance and maintenance of the metamodel definition.

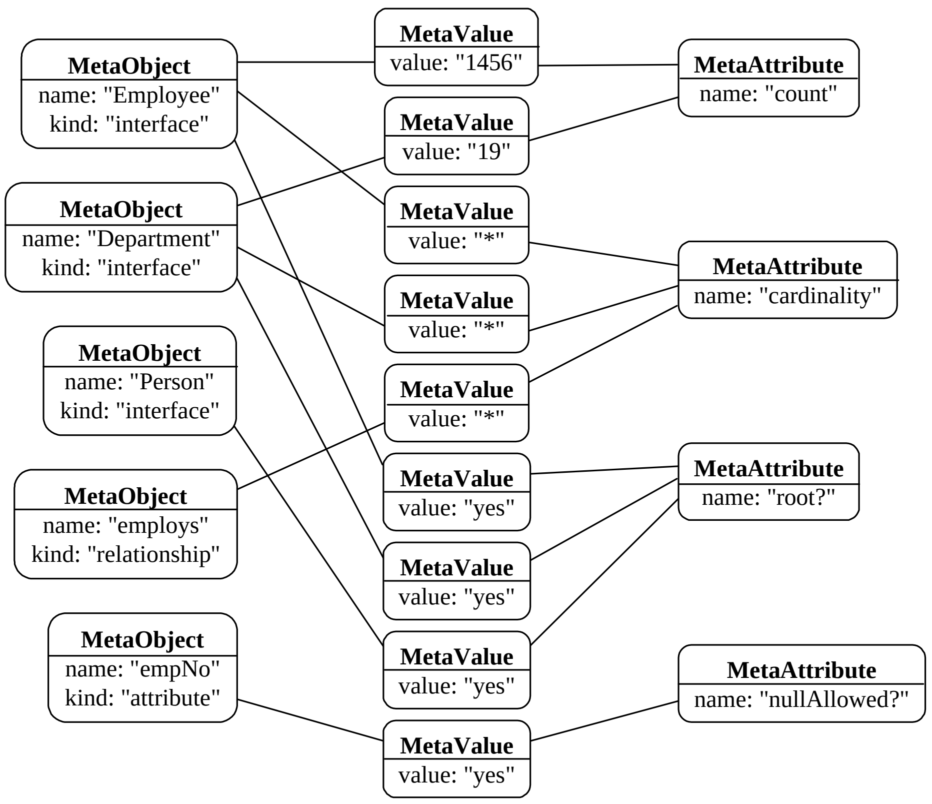


Fig. 9. A metamodel instance: the usage of meta-attributes

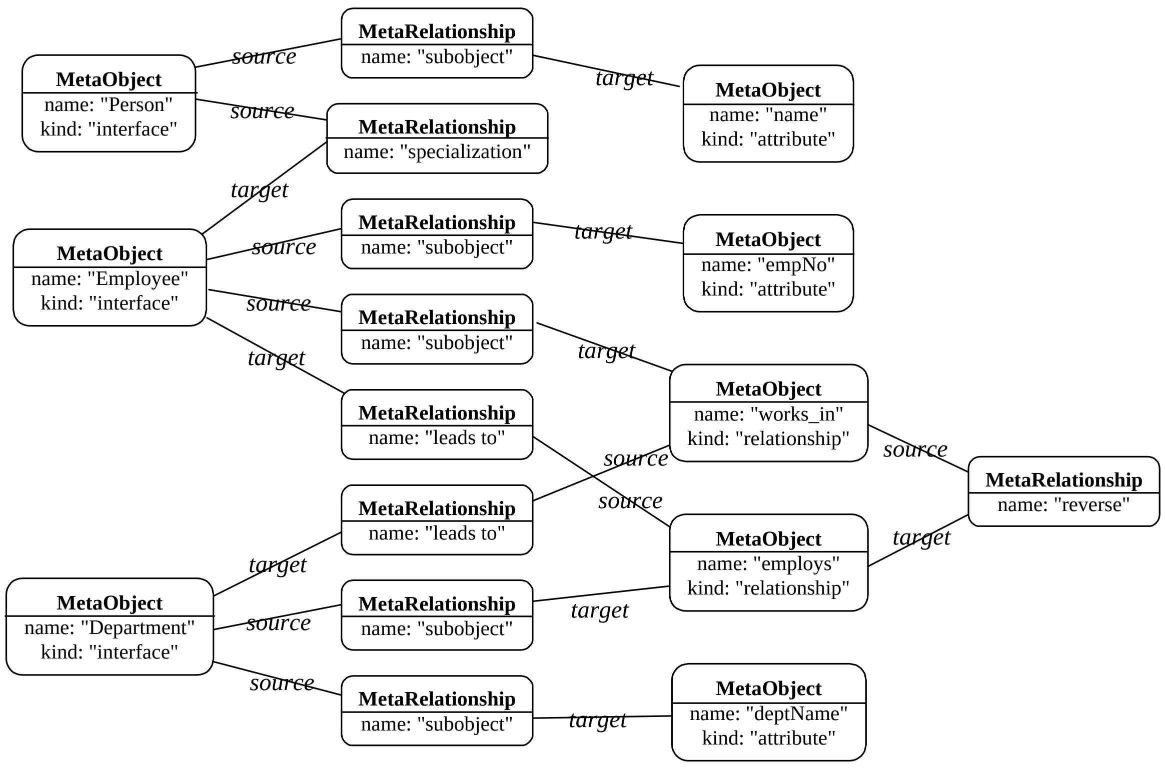


Fig. 10. A metamodel instance: the usage of meta-relationships

4.3 Conceptual view of the metamodel

As already mentioned, the required simplicity of database schema structure is contradictory to the descriptive role of metamodel, which emphasizes the need of metamodel expressiveness. Thus the conceptual view of discussed metamodel constructs is presented in a style of the UML metamodel. This form is used here for descriptive purposes only. The way of transforming such structure into the flattened form used during implementation is provided.

The base for metamodel definition

The style of metalevel definition chosen here, and represented e.g. by the OMG UML and MOF (Meta Object Facility) standards, follows the common four-level approach to metamodeling (see e.g. [16],[39],[41]), where the entities constituting a system are categorized into four layers: user data, model, metamodel and meta-metamodel. The user data are structured according to the definition provided by a model, and the model is defined in terms of a metamodel etc. The meta-metamodel is intended to be the minimum set of intuitive constructs (having a direct mapping to the implementation structures), that are used to define a metamodel. Such a multi-level metamodel definition, although inherently too complex to be directly implemented, allows to declare and discuss the introduced constructs in a clear way. The basic role of fourth layer (that is – a meta-metamodel) is to describe constructs used for the metamodel definition as well as to define means of extending the metamodel definition. However, in case of a DBMS metamodel, both issues seem to be of smaller importance because of following reasons:

- Unlike in a modeling language, where the model constructs can be perceived as volatile, the metamodel of DBMS materializes them in two ways. The metamodel has to be related to the storage model (see Fig. 11) in order to define relationships between regular data and metadata. Secondly, metamodel constructs have also to be hardwired into the database query language definition.
- Since a DBMS needs appropriate implementation of each introduced metamodel construct, the ability for a final user to extend the provided data model would be either very limited or very costly in terms of introduced complexity.

Based on the above arguments, it may be assumed, that the metamodel (M2 in Fig. 11), together with its mapping to the storage model, must be integrated into the DBMS implementation. The absence of an explicit definition of the meta-metamodel layer does not exclude the ability to extend its contents, especially because the flattened form if well suited for any extensions. However, the abovementioned limitations hold, making the more significant metamodel extensions the domain of DBMS vendors rather. The mentioned mapping to the storage model must determine (Fig. 11):

- How a given kind of metadata (e.g. a *Type* metaobject named "Employee") would be represented in the object storage (e.g. Composite Object). In Fig. 11 this mapping is shown using the «representation» dependency that should be understood as: "every metaobject from model (that is M1 entity) describing a *type* will be stored as a ***Composite Object***".
- How the instances of a given kind of metadata (e.g. an object of type *Employee* and its attribute *Name* having the value "Smith") would be represented (appropriately: Composite Object, and Primitive Object). This mapping is denoted in Fig. 11 by the «instance's representation» dependency that says: "every regular data element (M0 level entity) being an instance of *Class* metaobject from model (M1 entity) will be

stored as a ***Composite Object***, while each regular data element being an instance of *Primitive Type* metaobject from model will be stored as a ***Primitive Object***”.

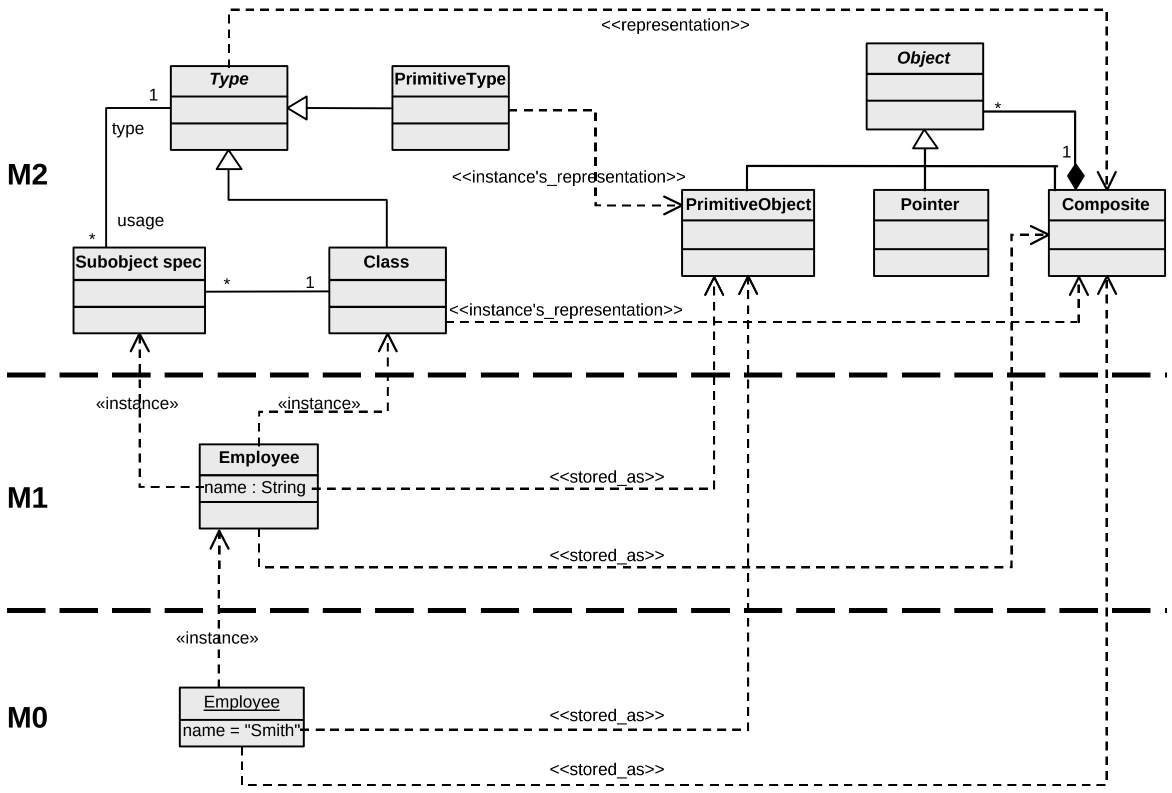


Fig. 11. The dependencies between metamodel (M2) and the object storage model

As can be seen, the «stored as» dependencies are derived from the combination of appropriate «instance of», «representation» and «instance’s representation» dependencies. The types of metadata, as well as the storage model primitives shown in this figure are exemplary only: they are chosen for their simplicity and they not necessarily fit the proposed metamodel and storage model assumed in this work (described in following sections). The main purpose here is to emphasize the connection between the metamodel and the storage model that also seem to conceptually belong to the M2 level.

That is, every newly introduced metadata element must be considered in terms of its relations to the storage model. For example the dynamic object roles mechanism (discussed in more detail later in this work), would require not only an extension to metamodel, but also a new, specialized element and link in the storage model to distinguish the relationship of being a role from regular association or composition among objects [28].

Metamodel core concepts

Fig. 12 shows an exemplary solution defining the core elements of the discussed metamodel. It is focused on the most essential elements of the object data model and, taking into account the different requirements concerning a database schema, it is by no means complete. Even with such reduced scope, the model becomes quite complex. As already stated however, this form makes it convenient to discuss some essential improvements introduced here and to compare them to the ODMG standard solutions.

All basic metamodel concepts inherit from the *MetaObject* and therefore possess the meta-attribute *name*, as practically every metadata element needs this property. The most important branches of this generalization graph are *Property*, which generalizes all the properties owned by *Interface*, and *Type* (described later), which describe any information on database object's structure and constraints. The *procedure* (method) definition is conventional. It allows for declaring parameters, events and return types (in case of a functional procedure). The parameter's *mutability* determines whether the attribute is passed as "input", "output" or "input-output". The use of meta-attribute *multiplicity* in the *StaticProperty* metaclass makes it possible to abstract from the definition of the collection concept.

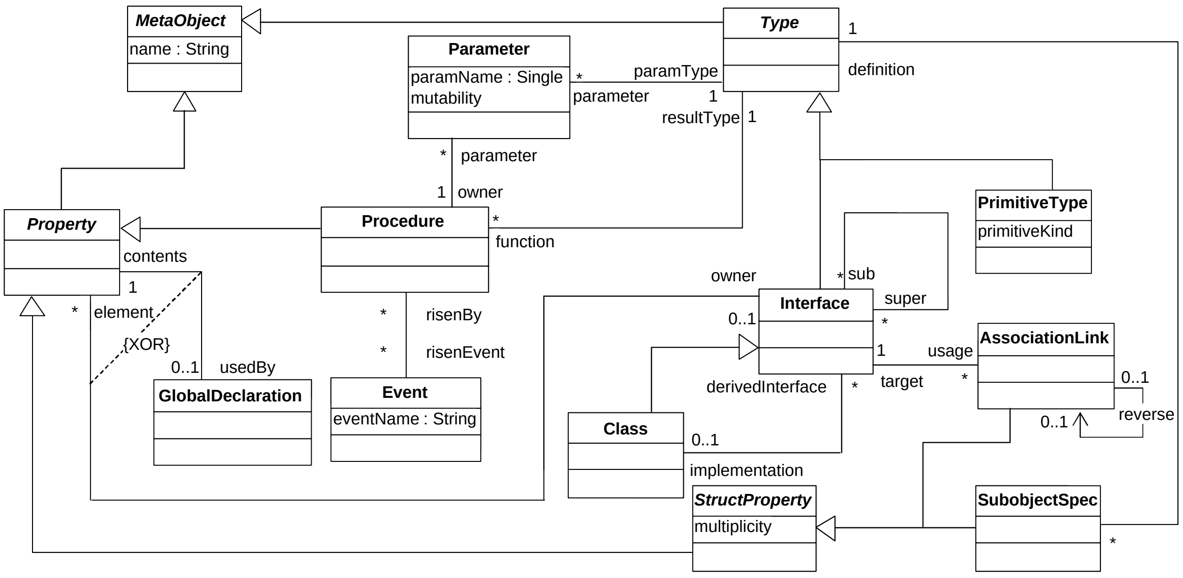


Fig. 12. Conceptual view of the proposed metamodel

Below, the most important features of the presented metamodel, especially those distinguishing it from the ODMG solution are enumerated.

- **Lack of method declarations**. In contrast to the ODMG definition and similarly to the UML metamodel, there are no method declarations in the proposed metamodel definition. As explained earlier, a generic query/programming language is suggested for the modification or retrieval of the schema information.
- **Object relativism**. For both the simplicity and flexibility of a DBMS it is desirable to treat complex and primitive objects in a uniform way. A *Type* concept, serving as a "common denominator" for both the complex objects' interfaces and primitive types has been introduced. Distinguishing *Subobject Link* from the *Association Link* allows for potentially arbitrarily nested object compositions.
- **Lack of the *extent* concept**. That concept, present in the ODMG specification seems to be rather problematic assuming object relativism with arbitrarily deep object compositions or when considering a distributed environment. Particular instances may have different meaning depending on their location within the data structure. Thus if every type declaration were connected with single, explicit extent, such situation would result in redundant type definitions. Moreover, if some instances exist as subobjects of different higher-level complex objects, separating them from their context and grouping within a single collection would be of no use. On the other hand, a distributed environment would make problematic performing the operations are checking the conditions that require access to all instances. For those reasons, in this work it is assumed that instances of declared types can be placed only within the separately declared locations.
- **Information on global variables declarations included in the schema as a separate construct**. For some purposes (e.g. the ownership and security management), the schema has to be aware of its instances. In absence of the *extent* concept, the declaration of root object entry has to be introduced as a separate construct. Note that within the metadata structure presented, such root declarations are treated almost identically as the subobject declarations provided within *Interface* definition.

- **No explicit collection types**. With the multiplicity declaration describing associations and sub-attribute declarations, the introduction of the collection concept into the metamodel can be considered redundant. The required properties of a collection can be described by the multiplicity attribute value of the *Static Property*. In some cases it would be also necessary to distinguish between the *list* and *multiset*. This can be accomplished by adding a meta-attribute "isOrdered".
- **Application of the terms "Interface", "Type" and "Class"**. *Type* can be described as a constraint concerning the externally visible structure of an object, as well as the context of its use. The role of an *interface* is to provide all the information necessary to properly handle a given object. Although the typing information remains the central component of an interface definition, the scope of the latter has to be much wider. In the presented metamodel it specifies public structural and behavioral properties, including raised events and possibly other properties. *Class* is an entity providing implementation for interfaces declared in a system. Every registered class contains all properties of a regular interface, which specify the complete list of features provided by the implementation. Every interface not being a class defines a subset of the features provided by its base class. In the other words, an *interface* narrows the *class's* default interface definition.
- **Bi-directional associations**. Although the reverse association of the *Association Link* is optional, which suggests that unidirectional links could occur, this concerns only the visibility of a given link through the *interface*. In order to support the maintaining of referential integrity, every created *Association Link* requires also a creation of reverse link.

Transforming conceptual metamodel into the flattened form

The conceptual view of metamodel like the one presented above needs to be transformed to the flattened form, which is more appropriate for direct implementation. In practice, this causes moving the majority of meta-metadata into the metadata level. The resulting schema (see Fig. 7) is not only very small in terms of its structure, but also it uses only the simplest concepts in its definition. This sub-section provides an overview of the implications of using such a simplified structure.

The process of mapping the metamodel structure like the one shown in the previous section can be described by the following rules:

- Every concrete entity from the conceptual view of a metamodel is reflected into the separate value of meta-attribute *kind* (see Fig. 7) of *MetaObject*.
- Inherited properties and constraints are imported into the set of features connected with a given value of *kind*.
- The meta-attribute *name* (required for every entity of the proposed metamodel) is mapped into the meta-attribute *name* of *MetaObject*.
- Every meta-attribute other than *name* is mapped into an instance of *MetaAttribute* in "flat" metamodel. All instances of *MetaObject* having an appropriate *kind* value are connected (through the *MetaValue* instance) to a single instance of *MetaAttribute* of a given name. *MetaValue* connects exactly one *MetaObject* with exactly one *MetaAttribute* used to describe that *MetaObject*.
- Every association existing in conceptual metamodel is reflected into the separate value of the meta-attribute *name* of *Meta Relationship*, and the second, other value, to provide the reverse relationship.[16]

It is now possible to summarize the meaning of the operations that can be performed on the flattened metamodel. Below the constructs are enumerated and the meaning of generic operations that can affect them is described.

- *MetaObject:*
  - Add / remove an instance (the combination of values of "name" and "kind" meta-attribute is unique among the meta-objects within a given scope) => schema modification;
  - Introduction of a new value of "kind" or its removal => change to the metamodel of a given tool;
  - Add / remove connected *MetaRelationship* instances => schema modification.
- *MetaAttribute:*
  - Add / remove an instance (the values of "name" are unique among MetaAttributes describing the MetaObjects of a given kind) => change to the metamodel.

[16] Note the difference in the nature between the meta-attribute "name" of MetaObject and the meta-attributes "name" of MetaAttribute and MetaRelationship. The former are the names defined for a given model, e.g. "Employee". The latter are determined by a metamodel, e.g "NoOfElements" or "InheritsFrom".

- *MetaRelationship:*
  - Add / remove an instance => schema modification;
  - Introduction of a new value of "name" or its removal => change to the metamodel.

As can be seen, due to moving the majority of meta-metadata elements into the metadata level, some of the operations identified above have more significant implications than just schema modification: they affect an established data model. Since with established DBMS, the majority of metamodel concepts must not be allowed to change, those operations are of rather limited use during the normal usage of a system. However, their straightforwardness makes it relatively easy to modify and extend the definition of the standard metamodel, that is, to realize "model tailoring" as phrased in [58].

Another important remark concerns the constraints connected with a given kind of metaobject. The metamodel form presented in Fig. 12 requires some well-formed rules that were not explicitly formulated even on that complex diagram. However, in case of the flattened metamodel, such additional constraints become critical, since practically no constraints (like e.g. multiplicities or the types of connected meta-entities) are contained in the metamodel structure. Therefore, in addition to the set of predefined values for meta-attributes like *kind* of *MetaObject* or *name* of *MetaAttribute* or *MetaRelationship*, the standard needs to define the constraints specific for each such value.

4.4 Database schema support for SCM

The main challenge of today's software development is dealing with complexity. In case of SCM this concerns especially the complexity of interdependencies among configuration items. Therefore, in order to better support the SCM aspect, the database metamodel definition should provide means to simplify the management of dependency information. There seem to be two general ways of achieving this objective:

- **Encapsulation / layered architecture**. Applying the encapsulation, both to narrow the interface of given classes, as well as to isolate the whole layers, allows to shorten the dependency paths.

- **Dependency tracking**. Even if the dependencies don't span across many configuration items, they still need to be recorded in a way that would guarantee completeness of such information and ability to easily extract it.

Both of these postulates are rather intuitive and are presently treated as a *sine qua non* in the development of large information systems. However, it is worth emphasizing, that dependency between applications and database schema constitute a special kind of dependency, which would be much more effectively handled when supported by the core DBMS mechanisms. As already stated, certain kinds of dependency information concern directly the database schema elements. Storing that information within the schema would not significantly complicate the metadata structure.[17] At the same time, it would advantageous, since the database dependency description would be centralized and, thanks to incorporating such mechanism into a database system, the dependency recording would be enforced by the DBMS itself.

Dependency kinds relevant to the metabase

This subsection revisits those of earlier enumerated dependency kinds that were considered relevant to the metabase, looking for optimum way of storing such information within DBMS schema.

Forward dependency and backward dependency

Since these kinds of dependency are mutually symmetrical, they could be registered using single construct, e.g. bi-directional association. Assuming traditional architecture, of special interest are the dependencies between external applications and the database, as well as dependencies among the database entities (the DBMS dependencies of other system elements, as the least critical, would not be tracked).

The target of the dependency association would be any (that is, behavioral or static) property of the database. The role of a dependent element would be played by either DBMS native procedure / method, or by an external application's procedure or module. Therefore, in addition to the regular database schema elements and dependency association, it is necessary to introduce a new construct, identifying an external procedure that the schema would be otherwise not aware of.

[17] Making the database aware of its dependent applications has previously been suggested e.g. in [13] through the concept of "application table". The intent of introducing that construct was slightly different though.

The optimum level of granularity of such information should be determined. The dependent element would be always a procedure / method. However, in case of external applications' dependencies, it could be practical to use a higher level, e.g. a whole application module. The target of a dependency can be either an interface or – assuming more detailed tracking – all its properties that a given routine accesses.

Side effect dependency

All requests to properties that are non-local for a given procedural unit or interface, can be qualified as side effect dependency. It is desirable to distinguish between passive and active side effects (the latter result in the database's state modification) and to include this information in the metabase.

Additionally, it is necessary to note, that when an interface definition is distinguished from the structure containing its instances, both the whole metamodel as well as the dependency tracking features, get more complex. The assumption that the user data definition is inseparable from the set of its instances is characteristic for the traditional relational model and contributes to its simplicity. However, as explained earlier, this approach poses some limitations thus it will not be followed here.

In that case it is not enough to connect the side effect dependency information with particular interfaces. Since the instances of a given interface can be stored in different globally accessible structures, the side effect dependency record should identify the database's global property declaration the manipulated properties are accessed through. For example one would like to know not only that a given procedure refers to objects of type *Product*, but also, that it operates e.g. only on objects stored within the global variable *avaliableProductsCatalog*. Therefore, in order to describe the side effect dependency it is necessary to identify the global variable a given procedure uses to begin its navigation, as well as all properties it then uses in order to get the reference to its target. Each such dependency concerning static property can be described as a read-only or updating. In case of database procedures, the analogous information would also be stored as a property of a given method in order to easily identify methods whose invocation does not change the database state.

Parametric dependency

This kind of dependency seems to be easier to handle than the side effect dependency, because here the dependent procedure does not have to be aware of the origin of provided parameter. No matter what kind of parameter it is: either the procedure reads, updates or returns newly created instance, it is only necessary to guarantee, that the definition of a given type stored in database metamodel has not been changed in a way that affect that procedure. In this case the target of dependency link would be simply a type definition the parameter refers to.

Proposed metamodel extensions

Providing features for storing the dependency information requires rather minor additions to the overall metamodel definition. This is thanks to the fact, that all constructs needed to describe the target of such dependencies are already part of the metamodel. Fig. 13 shows the necessary constructs added to the fragment of the metamodel from Fig. 12.

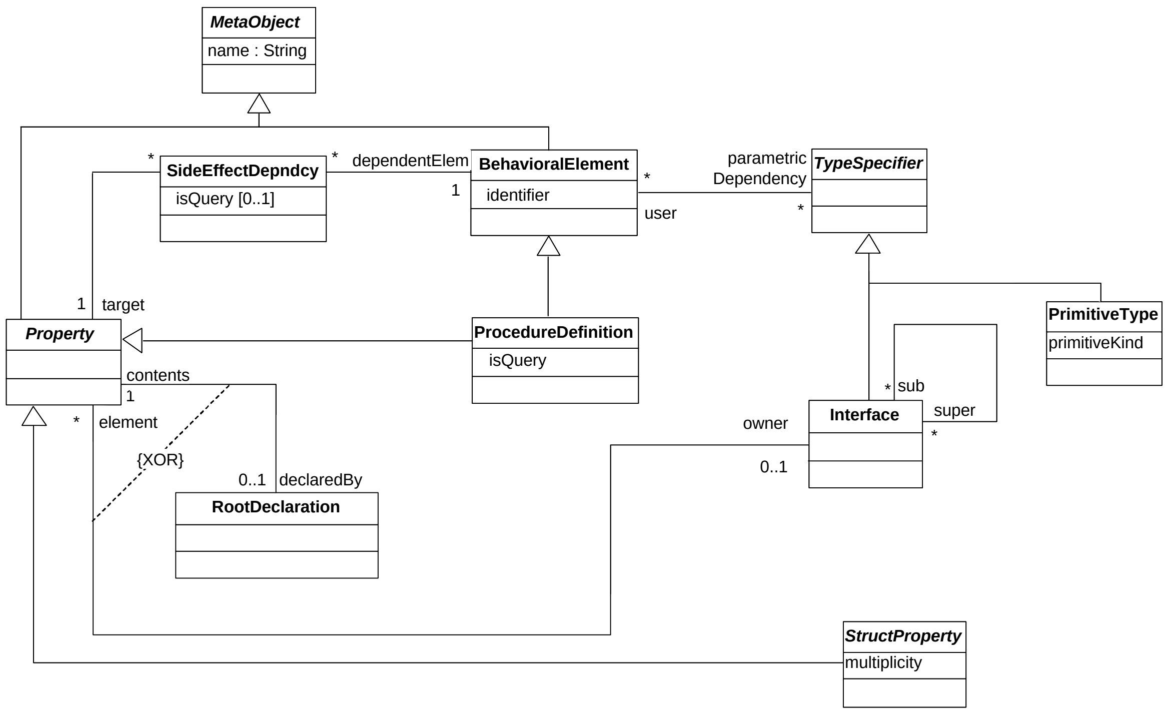


Fig. 13. A fragment of the proposed metamodel, with the dependency management constructs included

In order to record also the dependencies of elements located outside the DBMS responsibility (external applications using the database), a *Behavioral Element* concept has been added as a generalization of the *Procedure Definition*. Both elements can be the source of dependency relationship: the former (more general) can denote an external application's elements, while the latter always refers to the native procedures stored within the DBMS.

As suggested, the side effect dependencies are recorded for all elements that are used in navigation or manipulated. For each such dependency a metaobject of *SideEffectDependency*, connecting the dependent element description with dependency target is created. The *isQuery* meta-attribute provides binary information on the character of dependency: either pure read / navigation (value *yes*) or possibility to modify a given element (value *no*). Note, that it is not necessary to record the exact path of navigation. It is enough to determine, whether any part of a given procedure refers to a given property or not. The *isQuery* value is optional, because when the dependency target is a procedure, this information is not applicable. Parametric dependencies descriptions refer to type definitions (that is, metaobjects representing primitive types or interfaces). Other important dependencies relevant to the metabase (e.g. event dependency and definitional dependency) also can be derived from the outlined metamodel structure.

Figure below (Fig. 14) presents an exemplary fragment of a schema, showing the side-effect dependency of an external procedure *UpdatePrices*. That method refers to a global variable *OfferedProducts*, which is capable of storing arbitrary number of objects described by the interface *Product*. Through this interface, it can modify the subobject (attribute) *Price* of type *Currency*. In contrast, the global variable *OfferedProducts* is never modified by this procedure (it is used only for navigation).

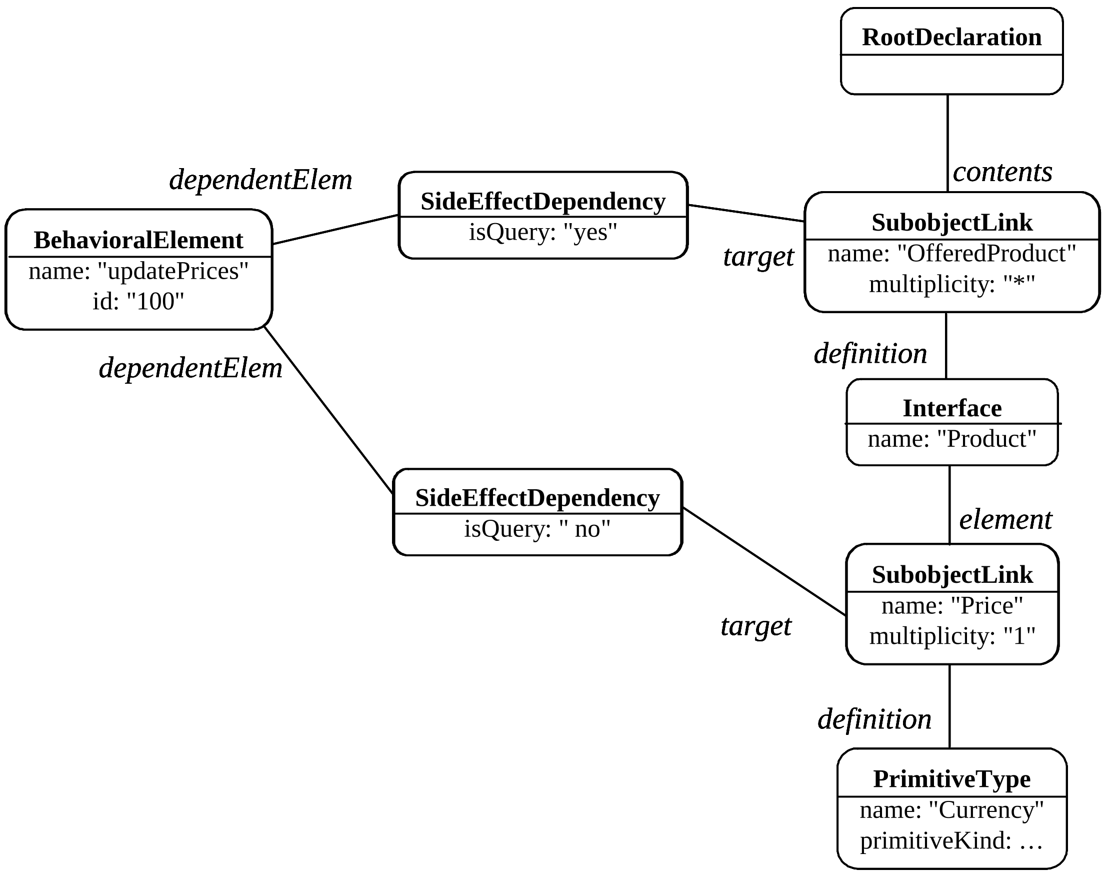


Fig. 14. An exemplary fragment of the DBMS schema, containing the side effect dependency information for an external procedure *updatePrices*

Collecting the dependency information

When recording the dependency information, it is necessary to face the problem similar to the one encountered between SCM system and other tools used in the software development: the development tools may lack the ability of providing the information needed for documenting the software configuration. Inability to automatically extract all the dependency information makes it necessary to manually document it, which can be considered as much less reliable. Thus almost all the advantage of storing it within the schema instead of SCM repository would be lost.

There are various options of realizing such automatic dependency analysis. The task is of reflective nature and solution depends on the way the database access is performed. For example, the DBMS could record the dependencies during one phase of the system testing and verify the completeness of collected information during further tests. For each recorded dependency it would need to receive some id of the procedure or unit that performs the database call. Note that adding that functionality to a DBMS that was not designed with such feature in mind could be problematic or at least result in a rather inconvenient usage scenario.

Such identification process should be separated from regular database security mechanism in order to not to deteriorate the DBMS performance during regular operation.

The functionality considered here also shows that the usage of metabase is not limited to internal DBMS implementation and that it thus should be realized as an externally available feature.

4.5 Extending ODBMS data model: dynamic object roles

As it was suggested in the previous chapter, the metamodel in the scope defined in the ODMG standard needs to be prepared for various future extensions, which can be subdivided into the following kinds:

- Object data model extensions towards more powerful and expressible constructs;
- Database feature extensions like active rules, stored procedures, views, security mechanisms etc.;
- Vendor-specific features not covered by standard;
- Limited means of extending metadata available for developers using a DBMS.

This section is intended to discuss the first from abovementioned kinds of extension. While it is impossible to anticipate the future evolution of the mainstream object model, there is at least one notion, namely – dynamic object role, whose importance and amount of related research makes it strongly desirable to consider in current proposals of an ODBMS metamodel. The notion is very intuitive and it is even featured as an object-oriented modeling construct equally fundamental as objects and associations among them [50].

Although very simple, the dynamic object role concept proves to be very useful for conceptual modeling, where it allows for more adequate representation of real-world dependencies, while avoiding the multiple-inheritance pitfalls.

The research in dynamic roles resulted in a large number of proposals assuming both object-oriented as well as other data models [3],[18],[59]. While presentation of those approaches is outside the scope of this work, looking at the problem from the database metamodel point of view requires formulating of the following remarks:

- Any proposal of such extension, more or less directly aimed towards standardization and commercial usage, needs to take into account the current state of the art. As far as it does not limit the clarity, usefulness or generally – the overall validity of proposed feature, its definition should refer to currently used solutions.
- The proposal concerning data model extension needs to be complete in the sense, it needs to provide both conceptual modeling constructs, as well as the ability to directly implement them. Both factors need to be considered when incorporating the concept into database metamodel.

For the abovementioned reasons, the following discussion of dynamic object roles is limited to the overview of related solutions that are currently broadly used and to the proposal of extending database metamodel with that construct in a way that keeps compatibility and semblance to already existing notions.

Dynamic and multiple inheritance problem

This subsection provides an overview of the most popular solutions and problems of multiple and / or dynamic inheritance modeling and implementation. In absence of more suitable constructs, these notions are often used to model simpler cases of statically of dynamically assigned multiple roles of an object. The popular examples are a person, who can at the same time be a student, an employee, a club member etc., being able to gain and lose such roles over time; a building serving different purposes etc. An example of a multiple though rather static classification can be a vehicle specialized according to terrain, powering system, its function etc.

Each of above examples can be described as a single object having different sets of properties connected with particular aspects of its existence. In case of a multiple yet static classification, the most straightforward of traditional approaches to address it would be a multiple inheritance. At the same time however, this solution bears the largest number of problems and limitations:

- Possibility of name conflicts among inherited features.

- Inability to dynamically reclassify object.
- Inability to describe more than one role of the same type (e.g. instance of a class *Employee* or *Student-Employee* can not store information on person's employment in two different companies).
- Combinatorial growth of the number of classes.

In case of singular inheritance the situation is even worse, since no code reuse occurs. That is, it becomes necessary to create "copy & paste" class definitions, where in addition the substitutability (e.g. between "Student-Employee" and "Student" class instances) can be lost.

The above remarks confirm that mixing different aspects of a given object's description into a single class is not a proper approach. The UML allows for specifying different specialization hierarchies for each criteria and even marking some of them dynamic. Such multidimensional classification results in much clearer model. However, this part of the UML specification is not very detailed and, what is the most important, implementation tools do not support such features, thus the model loses its clarity during implementation.

Another notion bearing some of the desired features are Java's inner classes. Such class is defined within the scope of its outer (base) class and its instances possess links to their base objects (being the instances of outer class). Thus such inner class can access properties of its base class in a way similar as if it were a subclass. Instances of inner class can be referenced either from inside of a base class's object (that is, being its attributes) or anywhere outside its base object (in the latter case making it unaware of its dependent instances). Thus the following properties, resembling the dynamic object role concept can be noted:

- Inner class can encapsulate some distinguished parts of object properties.
- Inner class's methods have access to the state and behavior of the base object as if they were defined in the outer class or its subclass.
- Arbitrary number of inner class instances connected with a given object can be created or removed during runtime.

- Instances of inner class connected with a given object of outer class are separated from each other, so no name conflict can occur.

As can be seen, the construct has many interesting features and can be useful in partitioning and encapsulating object's properties. However, this notion is quite far from widely agreed features of dynamic object roles in the following terms:

- There are no means of reassigning inner class object to be connected with another outer class's instance.
- Creating inner class definition requires access to the code of the outer (base) class. This breaks the "Open-Closed" principle concerning class design.
- Private properties of outer class are accessible to inner class's methods, which is probably undesirable in case of object role.
- The inner class's access to the properties of outer class does not provide the former with interface of the outer class. Thus there is no substitutability between outer and inner class's instances.

Other interesting features come from the aspect-oriented programming (AOP) language extensions. As mentioned in chapter 2, two general kinds of extending features are available:

- **"Pointcut" and "advice" declarations** modify certain elements of program's control flow by augmenting or bypassing specified kinds of statements.
- **Introduction** is capable to substantially change the definition of a given class, both in terms of behavior (new or overridden methods) as well as the structure (additional attributes).

The latter feature allows to isolate properties introduced to a class design on behalf of certain aspect or role of object's existence. However, it is necessary to note, that this mechanism is purely static and, although very powerful and advantageous in terms of maintenance, it is not suitable for implementing dynamic object roles. Its capabilities in this area seem to be analogous to static multiple inheritance.

The above overview suggests that although there are a number of already adapted similar solutions, none of them can satisfactorily address the issue of dynamic object roles, which are currently implemented in an indirect and rather low-level way (see e.g.

[15]). Considering the fundamental importance of that notion, it is possible to conclude that the dynamic roles should be introduced as a separate new construct both into modeling and into implementation languages and the DBMS area.

Features of dynamic object role

As suggested, dynamic object roles are usually considered as a mean to eliminate the shortcomings of inheritance mechanism provided by common OO programming languages and DBMS. Modeling notations often provide quite sophisticated kinds of generalization / specialization relationship (e.g. multi-aspect specialization, dynamic classification etc.), while implementation tools are usually limited to static, multiple or even singular inheritance. Moreover, even the means existing in modeling languages (e.g. UML[18]) are not exhaustive or not defined precisely enough to satisfactorily describe dynamically changing roles of objects. Thus the dynamic object role concept needs to be treated as a new quality both in modeling and implementation area, since analogies to other adopted solutions are quite superficial and may be misleading.

The simplest motivation behind that concept can be formulated as introducing a highly intuitive notion, capable to cover features of multiple, multi-aspect and repetitive inheritance while supporting its temporality. More thorough overview of the features of role mechanism that have been suggested in literature, provided in [50], enumerates the following properties:

- A role possesses its own properties and behavior, thus can be treated as a type.
- Different roles may share structure and behavior. This means, the role can make available the properties defined in its base role or base object, by means of inheritance or delegation.
- Roles depend on relationships. That is, model defines a pattern, under which object of certain type can be connected to other objects within the pattern, and this results in its playing a given role.
- An object may play different roles simultaneously. This makes the role mechanism a mean of a multiple classification.

[18] The UML standard has not defined support for dynamic object roles so far. The role concept exists in the specification in a number of other meanings, notably – the named association end, as well as a slot in collaboration specification (see [49]).

- An object may play the same role several times, simultaneously. This reaches further than a multiple classification since it assumes that an object can possess more than one set of properties defined by a given role.
- An object may acquire and abandon roles dynamically and the sequence in which roles may be acquired and relinquished can be subject to restrictions.
- Objects of unrelated types can play the same role. This seem to be natural, however it may be little problematic to assure consistency if a role implementation is assumed to be dependent on properties of its base object.
- Roles can play roles. This again suggests the role to have similar nature as a base object, as it may be viewed as such by some other role.
- A role can be transferred from one object to another. This bears semblance to UML's [41] composition relationship, where although the component is dependent on composition, it can be transferred to another one. On the other hand, an assumption that a given role can for a certain time remain unconnected to any object (like e.g. vacant position of a department manager), may be problematic (as above) concerning dependencies on base object properties.
- The state of an object and its features can be role-specific. This assumes multiple views of particular object, dependent of a selected role. This probably should assume also possibility of overriding behavior in a way analogous to traditional inheritance.
- Roles restrict access. Properties provided by the roles other than the currently accessed one are invisible. However, if access restricting assumes also hiding the properties of base object, this requires reconsidering of substitutability principle connected with traditional inheritance.

Although some of those properties seem to be to some extent contradictory, the above summary constitute a quite clear specification of features commonly expected from postulated dynamic object role mechanism.

Incorporating dynamic object role into the metamodel

This subsection describes an attempt to introduce dynamic object role concept into conceptual view of previously sketched object database metamodel. The main assumption is that although it needs to be a separate new construct, the new notion should be as far as possible "tuned" to existing data model concepts and mechanisms that support them.

Since role should be able to complement all kinds of properties the base object possesses (e.g. subobjects (attributes), association links, behavior (operations), as well as the ability to have its own roles), this requires supporting it with a specification similar to that of a regular object. That is, an interface specifying role properties and a class implementing them are necessary. Additionally, the role type definition brings an additional constraint: role needs to be connected with its "player", which can be a regular object or another role. Moreover, if a role is treated as a property a base object is aware of, a role multiplicity (like in case of attribute or association link), as well as applicability to exactly one base interface can be considered.[19] Following this path requires very little change it the proposed metamodel (Fig. 15).

Role interface becomes a special kind of *Interface*, distinguished by the fact that its instances, similarly like subobjects and association links are not independent (cannot be instantiated in separation from their base objects).[20]

An important question concerning constraints imposed on the dynamic object role construct is if a schema should specify (and allow) exactly one interface as a type of a base object for each role specification. This constraint seems to be justified by the following reasons:

- A role is indeed usually defined to extend the properties of exactly one object type. If there are more related types, they should have something in common, so generalization could be used as a role's base.
- Operations provided by a role interface can have their implementation dependent on particular properties (operations, attributes and association links) of the base object.
- If a reference to a role is intended to support all the base object's features, this flavor of substitutability requires appropriate type constraint.

[19] It would be possible to go even further, by providing labels for partitioning object's roles of the same type into distinct sets (e.g. person's *employee* role partitioned into full-time and part-time). However, such feature does not seem to be worth of additional complexity.
[20] Although the metamodel diagram (see Fig. 12 for a whole picture) allows all properties to be global (through the *root* declaration), in case of roles this should be excluded by an additional constraint.

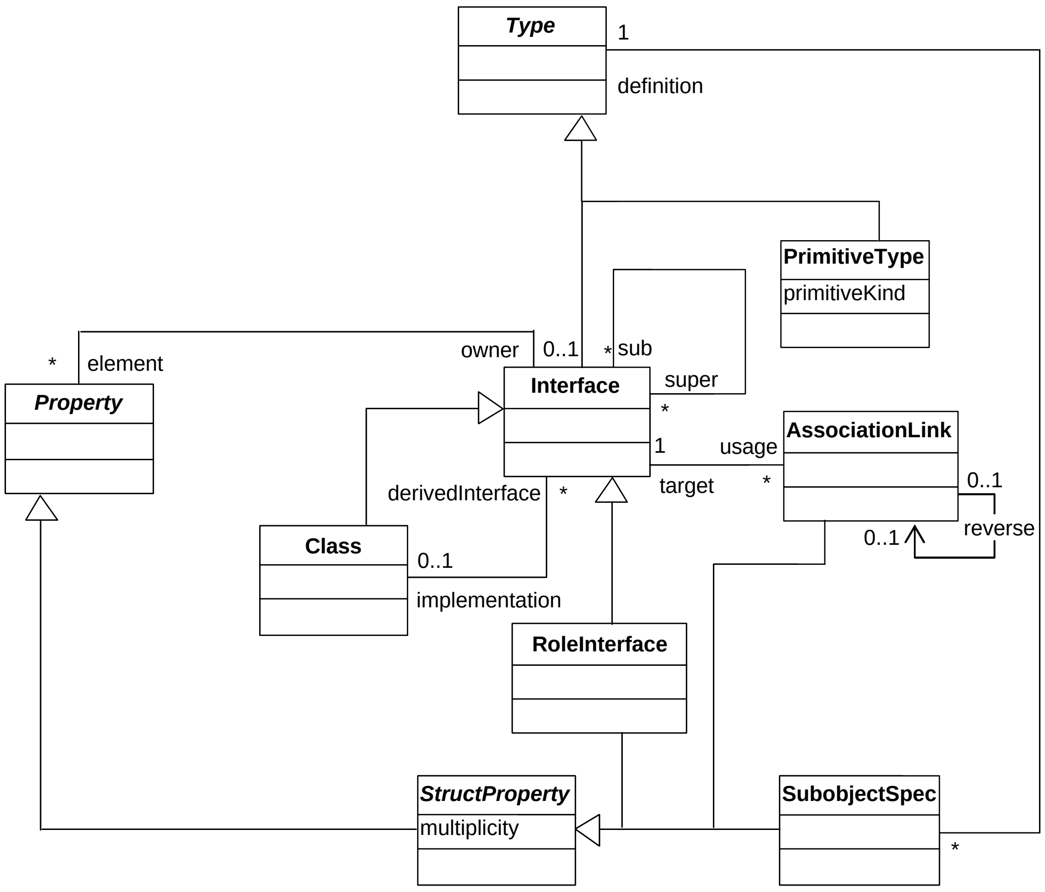


Fig. 15. Role concept incorporated into conceptual metamodel (updated fragment of Fig. 12)

On the other hand, also more “loosely” connected roles can be considered. Such roles would make no assumptions on their base objects’ types and serve as a “handles” or “labels” for a number of otherwise unrelated objects. In such a case it is possible to go even further by allowing the instances of such “loosely connected” role interface to occur either as roles or as separate objects. That solution would effectively allow the following kinds of interfaces:

- Interfaces describing base objects – which can be instantiated independently;
- Independent role interfaces, whose instances can occur either as separate objects or as roles loosely connected with their base, which in turn may be constrained to be an instance of one of specific types (indicated in the role’s definition);
- Dependent role interfaces, whose implementation does not depend on any base object’s properties, but the design constraints them to be always connected with a base of one of specific types;

- Dependent role interfaces, whose implementation requires access to a base object/role and which thus can be instantiated only as roles connected with their bases (of particular type).

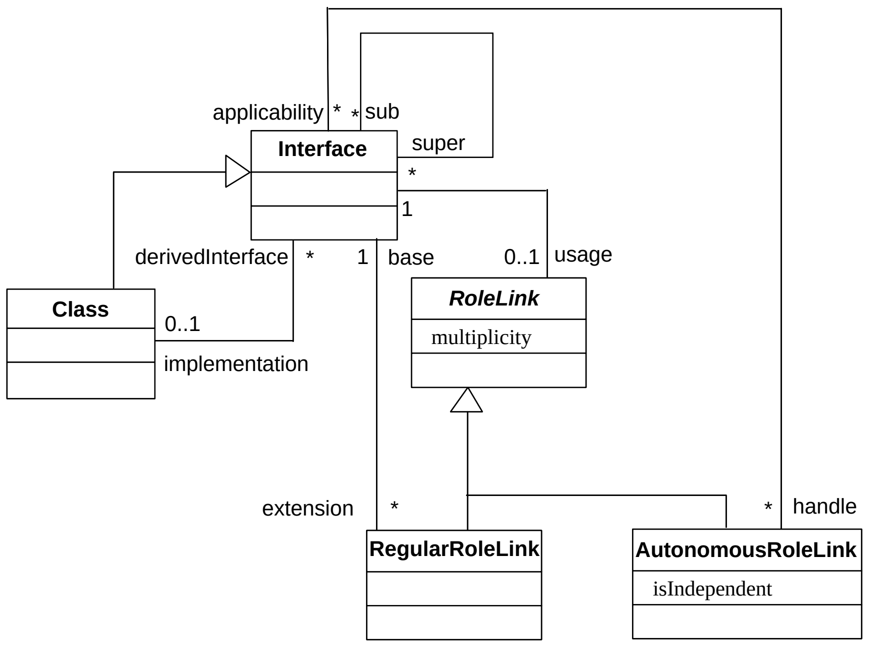


Fig. 16. Fragment of the proposed metamodel, introducing two kinds of dynamic object role definitions

This makes necessary to distinguish two kinds of role declaration. The first kind of role definition would depend on its base object's (or base role's) interface specification. Thus it needs to be specified as belonging to a given base interface, as assumed in Fig. 15. Another kind of role would resemble regular object in the sense that it does not require any base element. However, its definition may explicitly specify an arbitrary number of interfaces as intended base types. A separate flag is necessary to indicate if instances of such role are allowed to exist as regular objects. To accommodate both kinds of roles, the proposed metamodel has been restructured, to identify the role definition through relationships rather than through the *interface* concept specialization. Fig. 16 presents the appropriate fragment of the metamodel diagram. Both kinds of objects (roles and regular objects) are described by interfaces. Interface's association to the *role link* determines kind of its instances. If no such connection exists, interface describes a regular object. Otherwise it defines either a regular role, designed as an extension of other (base) interface, or autonomous role. In the latter case the *isIndependent* attribute determines if such role can exist as a regular object or if it is required to be connected to a base. From a point of view of a base interface, such role was named a *handle*, as it is connected to a base object to serve certain purposes, without requesting any of its specific properties.

As can be seen, this effectively results in three kinds of role dependency[21] that need to be indicated in design, including class diagrams. Fig. 17 presents all those kinds of role dependency, introducing an ad-hoc UML-based notation to distinguish them. Note the existence of role multiplicity specifications. Example a) presents the strongest form of dependency, where role implementation depends on properties of its base (*employee* role must always be connected with a *person* object). All role dependencies are denoted by filled triangle, to exploit analogies to UML's generalization and composition relationships. Example b) assumes that a role is implemented as self-sufficient and is allowed to exist as separated from its base (*dept manager* roles can be referenced in a system, although they may denote a vacant position). This loose dependency is denoted by dashed line. Finally, example c) is a more constrained form of the case b). Here, although a role is also implemented as independent on any base, it is required to always be connected with a base, which instantiates one of specified interfaces (*assignable resource* has to describe either a *car* or a *conference room*). The {XOR} constraint enforces it.

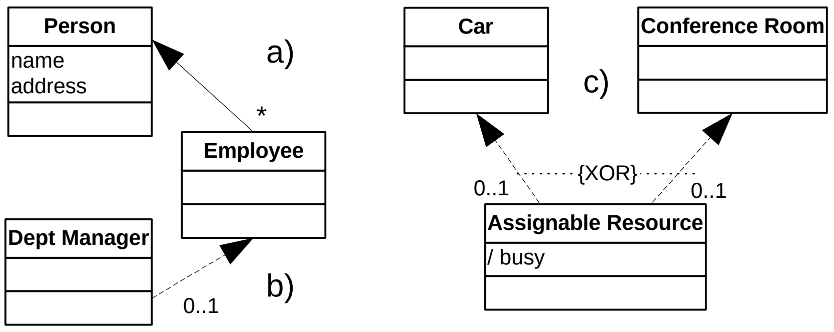


Fig. 17. Examples of different kinds of role dependency, introducing exemplary notation to distinguish them

Because in case of such autonomous role the constraint specifying to which base interfaces it is applicable does not seem to be essential, a significant simplification of

[21] The meaning of the word *dependency* can be twofold. Firstly, role's implementation can be dependent on the interface defined for its base object. Secondly, in the objects layer the role is connected with its base object or base role.

the metamodel can be achieved by removing that specification. Then the role-specific metamodel elements would be limited to just one recursive association (or association-class) over the *Interface* (similar to the one specifying static generalization-specialization graph), as shown in Fig. 18. The regular (dependent) role interfaces would be distinguished by the existence of the link to its base. The autonomous role interfaces would not differ from regular object interfaces. Thus all non-abstract interfaces except the dependent ones, would be allowed to form arbitrary combinations using the "base-role" link.

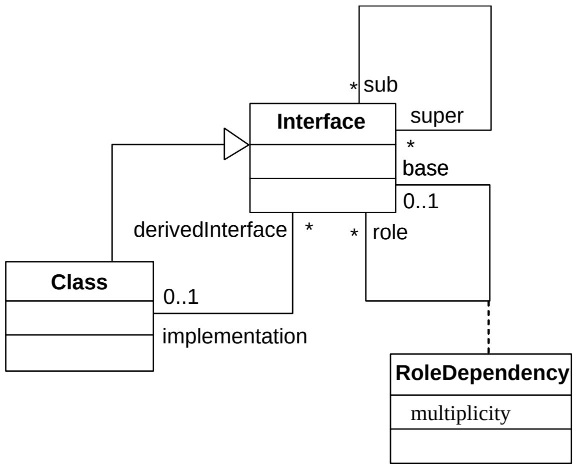


Fig. 18. Simplified solution of introducing the dynamic role concept into the metamodel

When a role extends or overrides the properties of its base object, a kind of substitutability known from traditional inheritance seem to be possible for language constructs dealing with roles. However, the analogy is quite superficial, and the link between role interface and its base interface requires in majority of cases substantially different treatment than regular generalization. Consider for example constraints on a regular generalization-specialization graph. Cycles are not allowed since the resulting structure would enforce mutually contradictory constraints on interfaces / classes within a cycle. Similar constraint is not obvious in case of a role dependency graph, if the weaker kind of dependency occurs. One could consider e.g. partial masking of object's properties in a given context by "covering" them with items coming from an instance of its subclass. An exemplary usage of such construction in metamodeling is presented in the next subsection (see Fig. 19). Although this example is rather peculiar, it is possible, that if no serious conceptual problems occur, such specific structure could be of use in conceptual modeling or as a design pattern.

Use of the role concept in meta-modeling

As mentioned, the dynamic role concept is assumed to be one of the fundamental abstractions of conceptual modeling. At the same time, the support for this original option in DBMS requires appropriate language extensions, as well as a specific primitive in the construction on database's object store (see [28]). Additionally, concerning previous postulate, that metadata should be manipulated in a way analogous to regular data, leads to a question, if this new notion could be of use in metamodel definition. This is not obvious, since for the sake of simplicity some constructs (like operations or nested object compositions) are intentionally not used in the presented proposal.

Dynamic object roles, although not essential for meta-modeling, seem to be well suited for describing some metamodel elements like e.g. class's derived interfaces (perhaps also in connection with some authorization/access control mechanisms). Appropriate modification of conceptual metamodel is presented in Fig. 19. This would allow instances of a given class to be viewed through different interfaces, modifying an original class's specification.

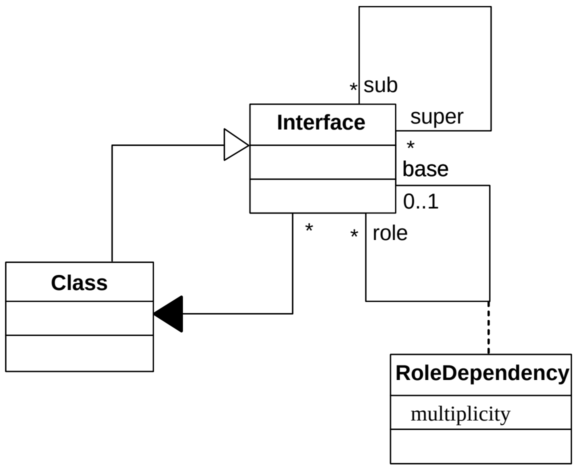


Fig. 19. Dynamic role mechanism used in the metamodel to define derived interfaces of a class

Concluding, dynamic object role is an example of a prominent notion, which is very likely to be introduced into future standard metamodel definitions. Since it constitutes an extension of a core data model, a number of different areas have to be considered. Among them are the following:

- conceptual modeling, including a graphical notation;

- consistent and preferably limited change to an original metamodel definition;
- DBMS query language constructs;
- Database object store model;
- Possible usage of newly introduced concept in modeling the DBMS metadata.

Introduced notion needs to take into account and adjust to pre-existing solutions, however only as far as such compliance does not appear to be a limiting factor.

4.6 Separation of concerns in a DBMS

The ability to modularize different concerns of developed system, discussed in chapter 2, should be also considered in the context of a database metamodel. The base requirements can be formulated as follows:

- Separate, single place for storing the code realizing a given requirement, which makes it reasonably easy to localize.
- Ability to connect the concrete implementation of a requirement in a way that does not affect the original functionality and is completely or partly transparent for it.

In some cases, an additional support from powerful reflective capabilities, as proposed for some programming languages, can be considered [6], in order to provide:

- The meta level interception (MLI) and the access to the processed environment using reasonably clear and simple programming constructs.
- Optionally, the ability to change the implementation of a given aspect during run time.

A possible solution, providing necessary modularization and flexibility, would be a usage of dynamic object roles in combination with an active rule mechanism, to encapsulate particular concerns (or aspects). Defining a rule within a role would allow to run an appropriate routine (after the interception of expected event), in the environment of the processed object. Moreover, with presence of introspective capabilities, the code would be generic that is, able to support the instances of a number of different classes.

However, it is necessary to note, that this would be a rather fine-grained mechanism, since roles need to be attached to particular instances and by their nature, are not shared. Thus it may be perceived to be redundant in case when particular requirement concerns all instances of a given class. This could suggest the need for a kind of class-scoped roles (analogous to static attributes and methods in object-oriented programming languages) or (a conceptually cleaner solution) – connecting a "concern-defining" role to a class rather than to an instance. However this would make a subject instance state not available directly, and necessary additional language elements and concepts would obscure the original idea.

4.7 Metamodel extensibility mechanisms

As already suggested, the extensibility of a database metamodel is important especially concerning future changes to the standard definition and for easy integration of vendor-specific features. Possible extensions made by DBMS users seem to be very limited and may concern rather some static extensions to incorporate custom metadata.

When considering database metamodel extension features, it is worth to investigate analogous solutions provided by the UML standard. Although the applications of these two metamodel definitions are fundamentally different, some conceptual similarities remain. One of them is the lack of operations used in a metamodel definition.

As mentioned in its overview, the UML metamodel provides three kinds of supporting features that can be used to extend the metamodel: constraints, tagged values and stereotypes (see p. 15).

Treating those features as a kind of checklist for a database metamodel, firstly it is necessary to mention the constraints. Indeed, the metamodel should be supported by means of formulating additional specific constraints over database objects. A query language seems to be very well suited for formulating such constraints. As already stated, a number of analogous constraints need to be implemented within a DBMS to maintain the consistency of the postulated flattened metadata structure.

Different roles the database metadata needs to fulfill, result in potentially numerous extensions augmenting the schema with additional metadata. For this kind of metamodel modifications, dynamic object roles can be useful, as they constitute a powerful mechanism that could easily realize the features of *tagged values* and *stereotype*.

5 Implementation

The prototype implementation of metadata repository presented here realizes the postulate of flattening the metamodel and is intended to prove its advantages in terms of simplicity and extensibility. Another issue investigated within this prototype is the DBMS schema support for the SCM through the tracking of database dependencies.

The implementation is of limited scope in a sense that it is not a part of a complete DBMS prototype. The most important consequence is that the postulate of using generic operations of a query language to access the metadata is not realized here.[22]

It was developed using pure Java language plus Objectivity/DB ODBMS as a persistence mechanism. To realize run-time software dependency recording, the AspectJ language extension [2] has been used.

The implementation consists of two main areas. The first is a generic GUI-based metadata repository. The repository is based on the proposed flat metadata structure. It allows for definition of arbitrary metamodel in terms of allowed combinations of metaobject kinds, meta-relationships and meta-attributes describing metaobjects of particular type. This task is realized using an application named "*Metamodel Manager*". After defining such a metamodel it is possible use another application – "*Model Manager*" – to create model, which is stored within the flattened structure and respects the constraints introduced by the metamodel.

The second area provides an example of flattened metamodel, applying the approach to the subset of Objectivity/DB schema structure. In this case metamodel is determined and model is extracted from specified, populated Objectivity database. Thus the GUI-based functionality is limited to a metadata viewer. Moreover, the Objectivity/DB metamodel has been extended to store the software dependency information, which in current implementation can be extracted automatically during application runtime (e.g. in application testing phase).

[22] Unfortunately, this fact makes the benefits of flattening the metamodel less significant. As will be shown the flat metamodel is value-oriented, which makes it less convenient to handle within the chosen implementation environment (ODBMS using Java binding). The tasks like object lookup based on values of its properties or checking the conditions requiring navigational access would be much easier to perform using a query language.

Implementation description presented here is structured as follows. Two parts distinguished above are presented separately. Each part starts with a summary of externally available functionality and a description of its user interface (if applicable). Later, the high-level description of implemented classes and their functionality is provided.

5.1 Flattened metamodel – conceptual view

The suggested structure of flattened metamodel has already been presented in the previous chapter. The diagram presented there has been included (with subtle change) in this section (Fig. 20) to describe the starting point of this implementation.

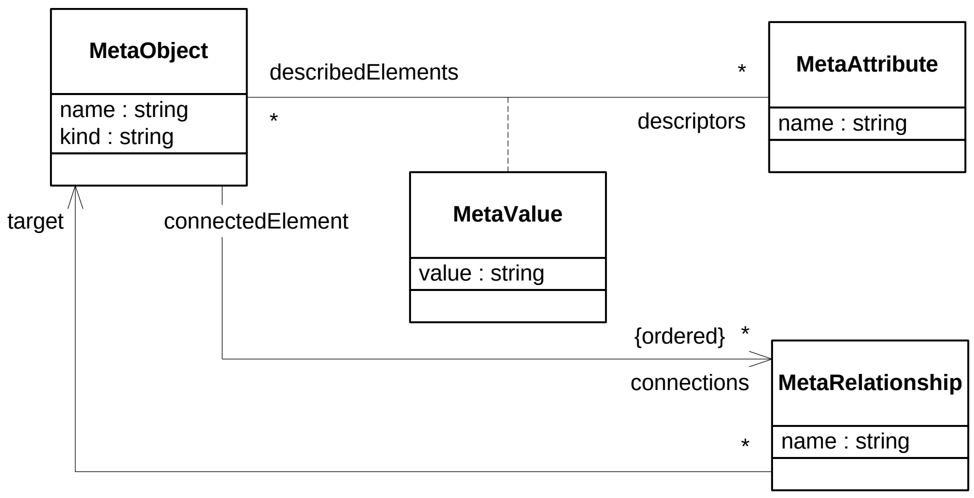


Fig. 20. Conceptual view of the flattened metamodel assumed during implementation

The only change introduced into this diagram is the usage of *association class* (see [41]) to denote meta-value. This notation indicates that the implementation does not support multivalued meta-attributes. Database design will provide the names of introduced metaobject. In contrast, the possible values of remaining attributes shown here (together with constraints on their combinations) come from a specific metamodel definition. Since the former activity depends on the latter, the "Metamodel Manager" functionality will be described first.

5.2 Metamodel manager

This application allows to define metamodel-specific properties that are not contained within the flat structure. This information supports the modeler's work by in the following ways:

- New metaobjects are given (at creation time) a set of properties (slots for the meta-attributes' values) according to meta-attributes the metamodel designer defined to describe the metaobject kind chosen.
- It is possible to select the kind of created metaobject only from the list of kinds defined in metamodel. The same holds for names of meta-relationships.
- Moreover, one can create only meta-relationships, whose origin and target metaobjects kinds have been allowed for a given meta-relationship.[23] Like previous, this constraint is supported by GUI, whose lists and combo-boxes used during metaobject edition show only elements that respect the defined constraints.
- Fore more specific constraints it is possible to designate a class with special interface, whose *validate(MetaObject)* method would check constraints specific to a given metaobject kind (e.g. checking the rule that meta-attribute *isAbstract* of metaobject *Class*[24] must have either a value "true" or "false" and that in the latter case none of metaobjects *Operation* connected to it may have the *isAbstract* attribute value set to "true") or to a given meta-relationship name (e.g. that the *specializes* meta-relationships do not create loops).

Thus, because with an empty metamodel definition, the following mechanism would prevent from creating any metaobject, the Metamodel Manager is the first step in the usage scenario of this software.

User interface and functionality of Metamodel Manager

Since the whole metamodel implementation presented here uses Objectivity/DB as a persistence layer, a *federated database* of that DBMS must be created first. The application requires to specify the path to a *federated database* before any other

[23] E.g. metamodel definition may define that the *generalizes* meta-relationship can connect meta-object of kind *Class* with metaobject of kind *Class* or metaobject of kind *Interface* with metaobject of kind *Interface*.
[24] That is, metaobject, whose *kind* equals "Class".

function can be invoked. Within the chosen *federated database* a *database* named *SchemaDB* is opened and created and within it the container *MetamodelCont* is accessed in an analogous way. If some elements of metamodel have previously been defined, they are read from the database, and the user interface elements are populated with them (Fig. 21).

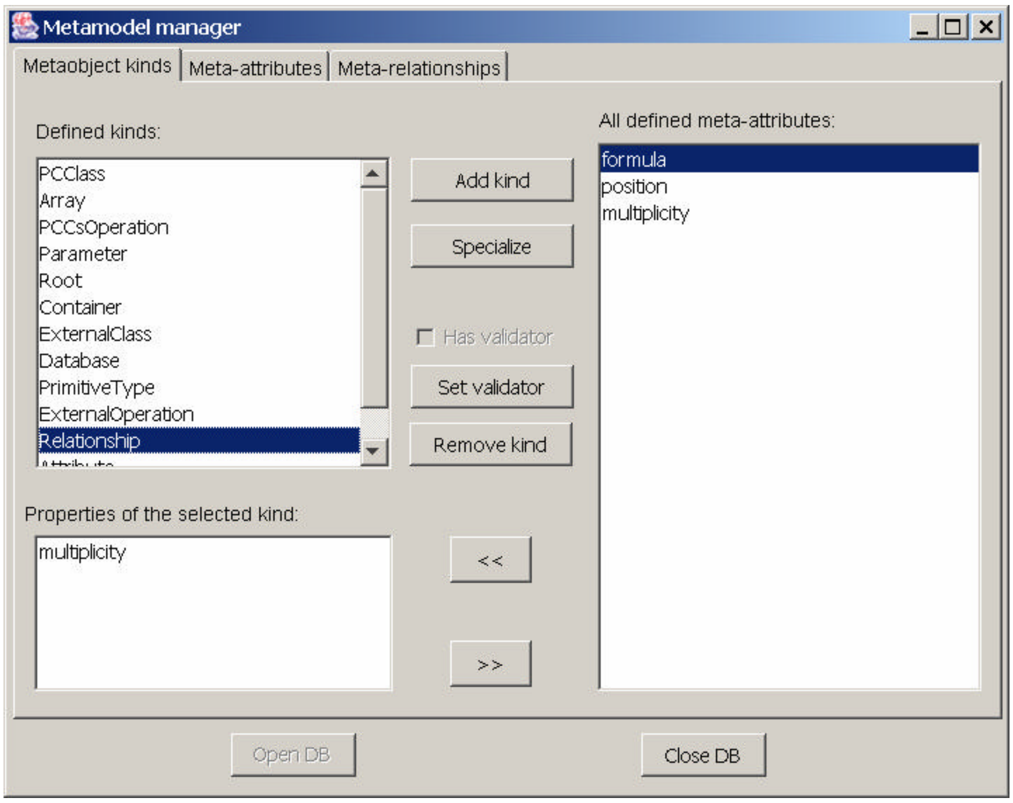


Fig. 21. Metamodel Manager window – the "Metaobject kinds" tag

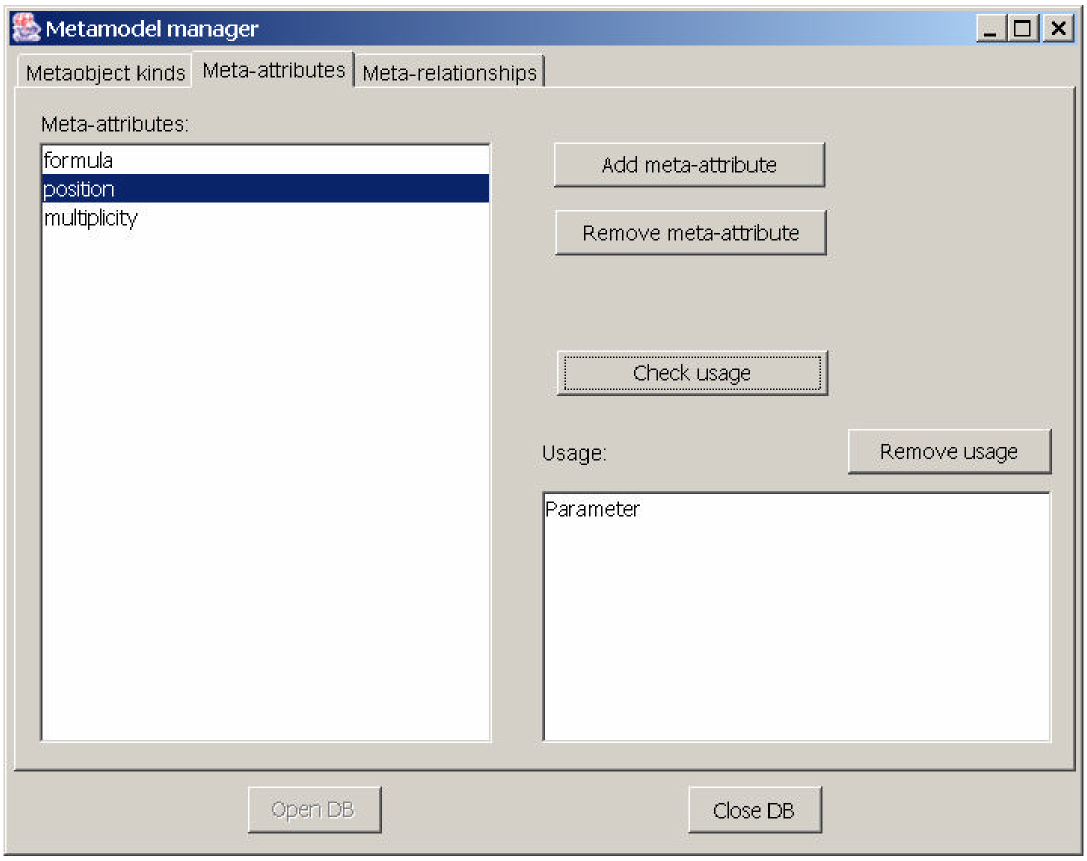
The application window consists of three tags. The first of them, called "Metaobject kinds" provides the following functionality:

- Creation of a new metaobject kind or removal of an existing metaobject kind (the latter is possible if a given kind has no instances).
- Creation of a new metaobject kind as a specialization of already defined kind. New metaobject kind is given all the properties (that is – assigned meta-attributes) that the chosen (prototype) kind possess. New allowed combinations[25] of meta-object kinds connected by meta-relationships are created, to let the new kind appear in all

[25] More precisely, it is a combination of meta-relationship name with an ordered pair of metaobject kind names. For brevity however, the less precise term "metaobject kind combination" will be used for the rest of this work.

meta-relationships that accept the prototype kind at the given end. For simplicity, the original kind is a prototype rather than generalization, since after the specializing kind is created, both kinds can be modified independently. In other words the generalization link is not maintained between two such kind definitions.

- Assignment of previously defined meta-attribute as a property describing the selected metaobject kind or removal of such meta-attribute from the list of properties of a given metaobject kind.
- Assignment of a validator class dedicated for checking specific constraints connected with a given metaobject kind. This is realized by providing a package name-qualified name of a Java class. Application checks if a class of the name provided is available and if it implements the *MetaObjectValidationMethod* interface.

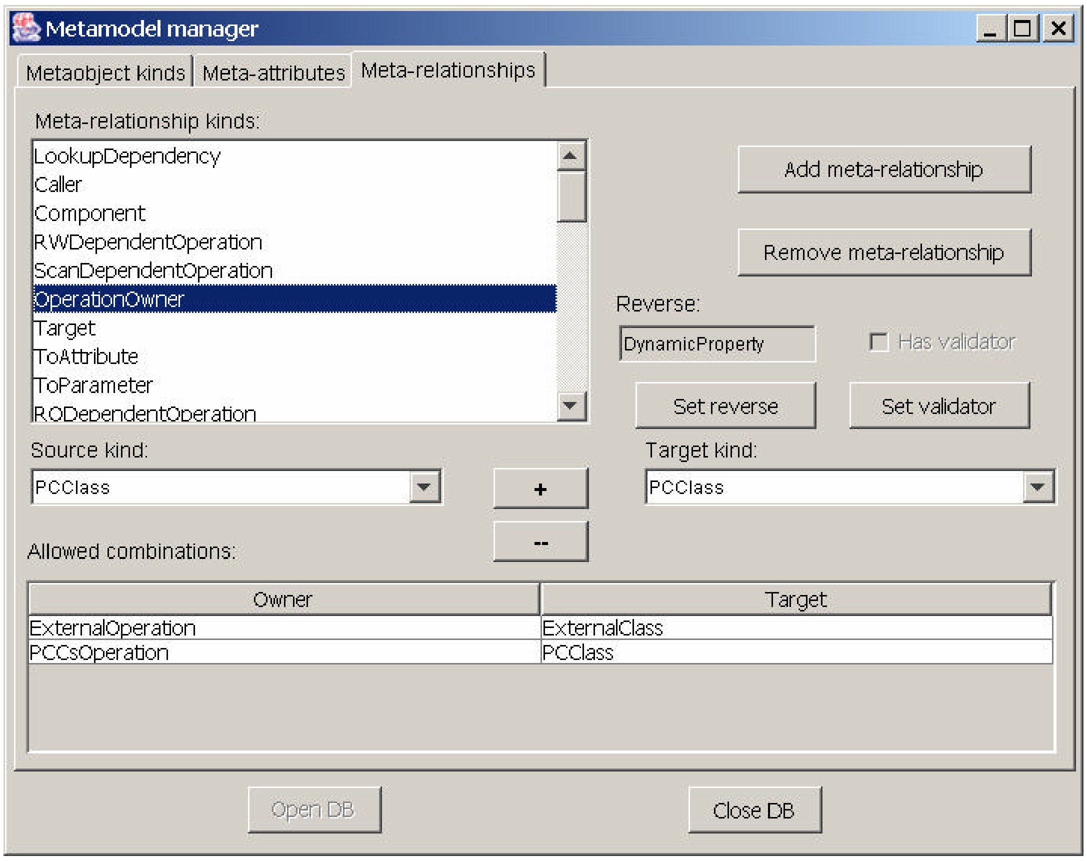


formula
position
multiplicity

Parameter

Fig. 22. Metamodel Manager window – the "Meta-attribute" tag

The next tab titled "Meta-attributes" (Fig. 22) allows for adding and removing meta-attributes. A meta-attribute cannot be removed if it is currently used as a property of one of more metaobject kinds. To check this, a manually invoked functionality lists the metaobject kinds a selected meta-attribute is used by. It is thus the reverse side of relationship between metamodel kind and meta-attribute used in the previous tab and it is added here for convenience. Instead of browsing all metaobject kinds in the first tab, one can quickly locate usage of selected meta-attribute and to remove it from the list of properties of referenced metaobject kinds.



Meta-relationship kinds:

- LookupDependency
- Caller
- Component
- RWDependentOperation
- ScanDependentOperation
- OperationOwner
- Target
- ToAttribute
- ToParameter
- RODependentOperation

Reverse: DynamicProperty

Source kind: PCClass

Target kind: PCClass

Allowed combinations:

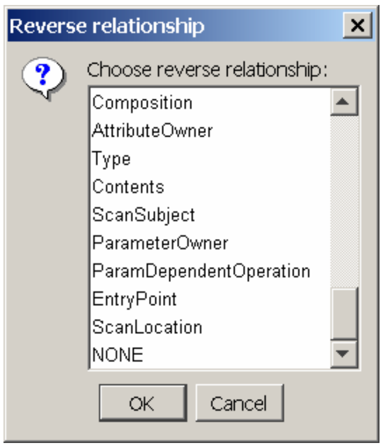
Owner	Target
ExternalOperation	ExternalClass
PCCsOperation	PCClass

Fig. 23. Metamodel Manager window – “Meta-relationships” tag

The last tag, called “Meta-relationships” (Fig. 23), is provided to manage the meta-relationship descriptions. It offers the following functionality:

- Addition / removal of meta-relationship name.[26] The latter will not be allowed if meta-relationships of a given name exist in the model.
- Addition / removal of metaobject kind source<->target pairs allowed for selected meta-relationship name.

[26] In fact this could be called *kind*, since a meta-relationship’s *name* is of the same nature as a *kind* of metaobject.


Reverse relationship

Choose reverse relationship:

Composition
AttributeOwner
Type
Contents
ScanSubject
ParameterOwner
ParamDependentOperation
EntryPoint
ScanLocation
NONE

Fig. 24. Reverse relationship name-choice dialog

- Pairing mutually reverse meta-relationship names (see Fig. 24; both meta-relationships names have to be defined earlier). In contrast to the specializing metaobject kind, which is creation time-only shortcut, the information on reverse meta-relationship names is kept and the allowed metaobject kind combinations for both of them are maintained accordingly to this paring.
- Setting validator class dedicated to a given meta-relationship name (analogously like in case of metaobject validator class. In this case the selected class has to implement the *MetaRelationshipValidationMethod* interface).

Probably it would be also useful to introduce one more tag that would show the allowed meta-relationship in a metaobject kind-centric way, thus showing the names of meta-relationship applicable to selected metaobject kind.

Implementation classes of Metamodel Manager

The user interface layer has been separated from the rest of the project. This is reflected in the fact that all GUI-related classes are located in a separate package called *ui.metamodel*. Since they represent a conventional usage of standard Java GUI library, their description is limited to the screenshots and associated summary located in the "user functionality" subsections of this chapter. The rest of the classes constituting the application are located in *metadataRep.metamodelManager* package.

The central non-UI class of Metamodel Manager is *MetamodelDictionary*. It is responsible for establishing and closing connection with selected *federated database* as well as for handling any requests concerning querying or modifying metamodel definition. For cooperating with Model Manager, the interface *MetamodelAdvisor* of *MetamodelDictionary* is used. The functionality of this interface is narrowed to read-only operations that are used to provide hints and validity checks during model edition (as described in the next section). It supports the following operations:

- Getting an array of defined metaobject kinds;
- Getting an array of metaobject kind pairs, describing the combinations allowed for selected meta relationship name (provided as a method parameter);
- Getting an array of meta-relationship names that accept the selected metaobject kind as an origin.
- Getting an array of metaobject kinds that are allowed as a target of meta-relationship of the selected name, coming from a meta-object of the selected kind;
- Refreshing the pool of defined metaobject validator and meta-relationship validator classes;
- Providing the database session object to allow Model Manager to access database after successful initialization.

Moreover, *MetamodelAdvisor* inherits from two other interfaces: *MetaObjectValidator* and *MetaRelationshipValidator*. Thus every validation request from metaobject or meta-relationship comes to *MetamodelDictionary* and is dispatched there: the kind of requesting metaobject or the name of requesting meta-relationship is checked, and validation method of the class assigned as its validator is invoked. If a given metaobject kind or meta-relationship has no validator, the result of validation is assumed to be successful and appropriate value is returned to the caller.

The rest of functionality, available only if explicitly referring *MetamodelDictionary*, consists of the following operations:

- Creating/removing meta-attribute names;
- Creating/removing metaobject kind definitions;
- Creating/removing meta-relationship definitions;
- Assigning/canceling selected meta-attribute as a property of selected metaobject kind;

- Getting an array of the names of meta-attributes being the properties of selected metaobject kind;
- Allowing/denying a selected pair of metaobject kind to be connected by selected meta-relationship; checking if a given combination is allowed;
- Cloning metaobject kind definition as a specialization of an existing one;
- Getting/setting the names of metaobject- and meta-relationship validation classes;
- Checking if a given metaobject kind is defined within metamodel;
- Checking if a given meta-relationship name is defined within metamodel;
- Checking if a meta-attribute of a given name is defined;
- Getting an array of names of metaobject kind using a meta-attribute of selected name;
- Paring mutually reverse meta-relationship names;
- Assigning a name of validator class for selected metaobject kind;
- Assigning a name of validator class for selected meta-relationship name.

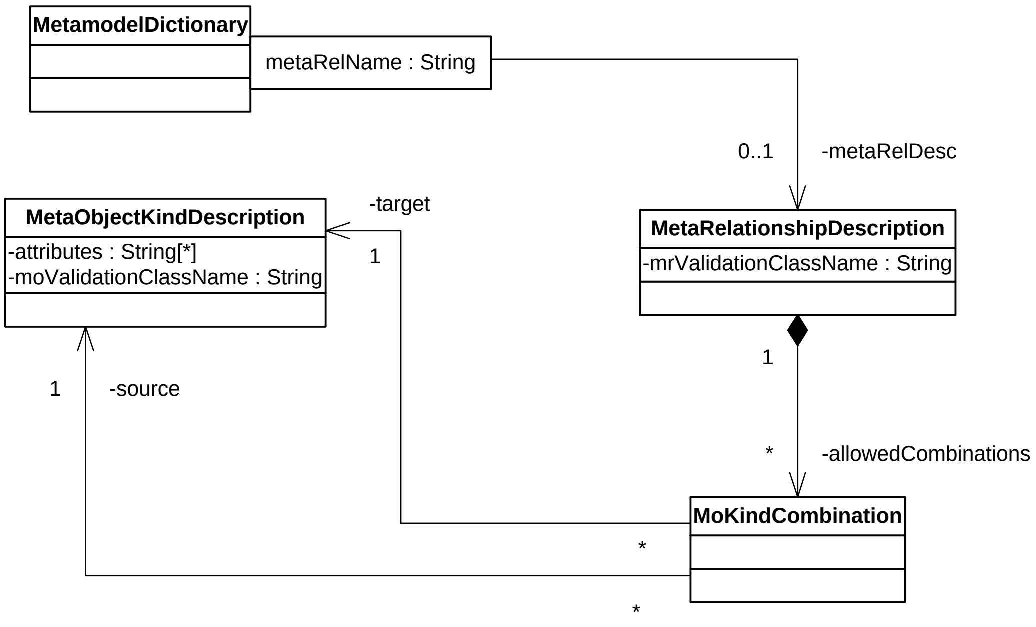


Fig. 25. Metamodel-defining classes for the implementation of the flattened metamodel (UML diagram)

Fig. 25 shows the class structure used to store data necessary for the above functionality. Although it is very simple, significant complexity of specific well-formedness rules may be hidden within the validating code. However, this problem may be effectively reduced in presence of a full-featured query language to manipulate metadata, as it would allow for a clear, declarative formulation of those constraints.

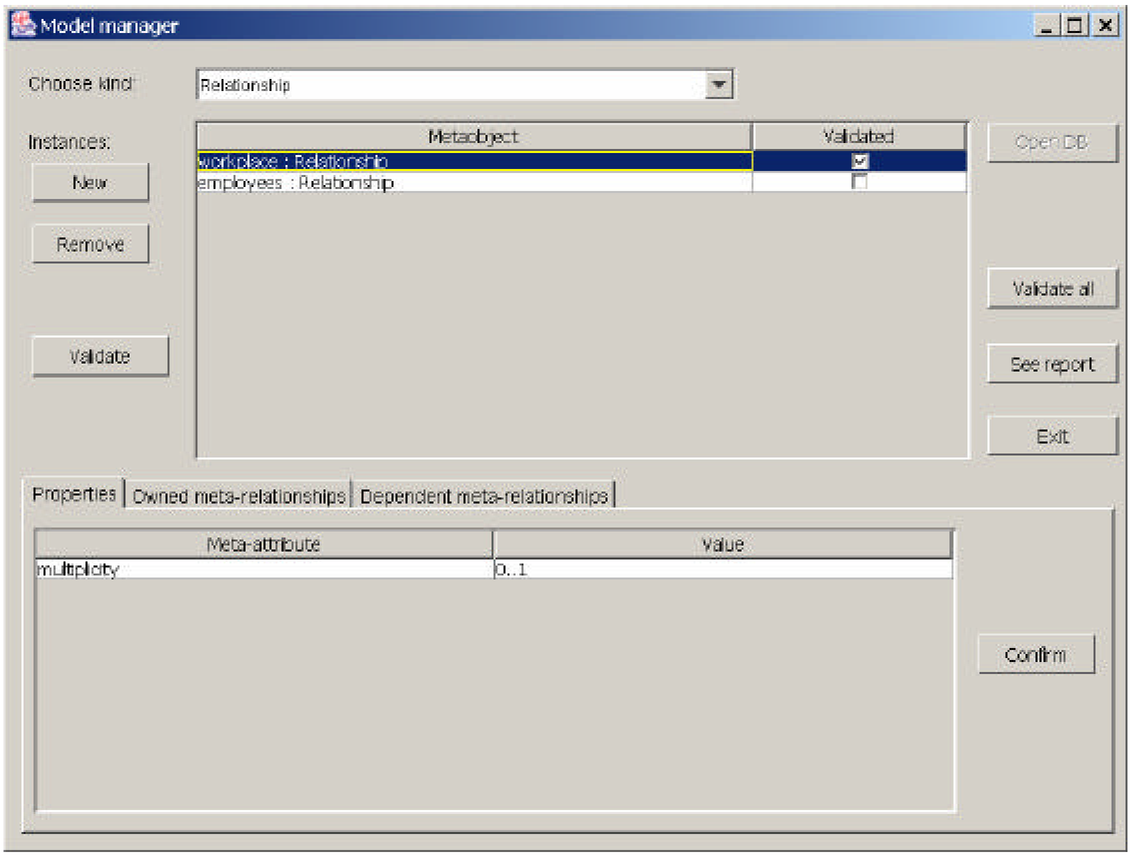
5.3 Model Manager

Having defined a metamodel makes it possible to use another application – *Model Manager*, to create model that would be stored within the flattened metamodel structure described at the beginning of this chapter. The application realizes the following functionality:

- Browsing/creating/removing the instances of the selected metaobject kind;
- Performing validation of selected metaobject, meta-relationship or the whole model;
- Browsing the validation report and navigation to the elements where inconsistencies were found;
- Updating the values of meta-attributes of selected metaobject;
- Creating or removing meta-relationships originating at selected object;
- Navigation along meta-relationships to other meta-objects.

User interface and functionality of Model Manager

To access model, one needs, like in case of Metamodel Manager, to specify the *federated database* to work with. Since the metadata-updating features of Model Manager require access to the read-only functionality of Metamodel Manager (through the abovementioned *MetamodelAdvisor* interface), the latter is also prepared during this initialization.



Metaobject	Validated
workplace : Relationship	✓
employees : Relationship	☐

Meta-attribute	Value
multiplicity	0..1

Fig. 26. Model Manager window – the “Properties” tag

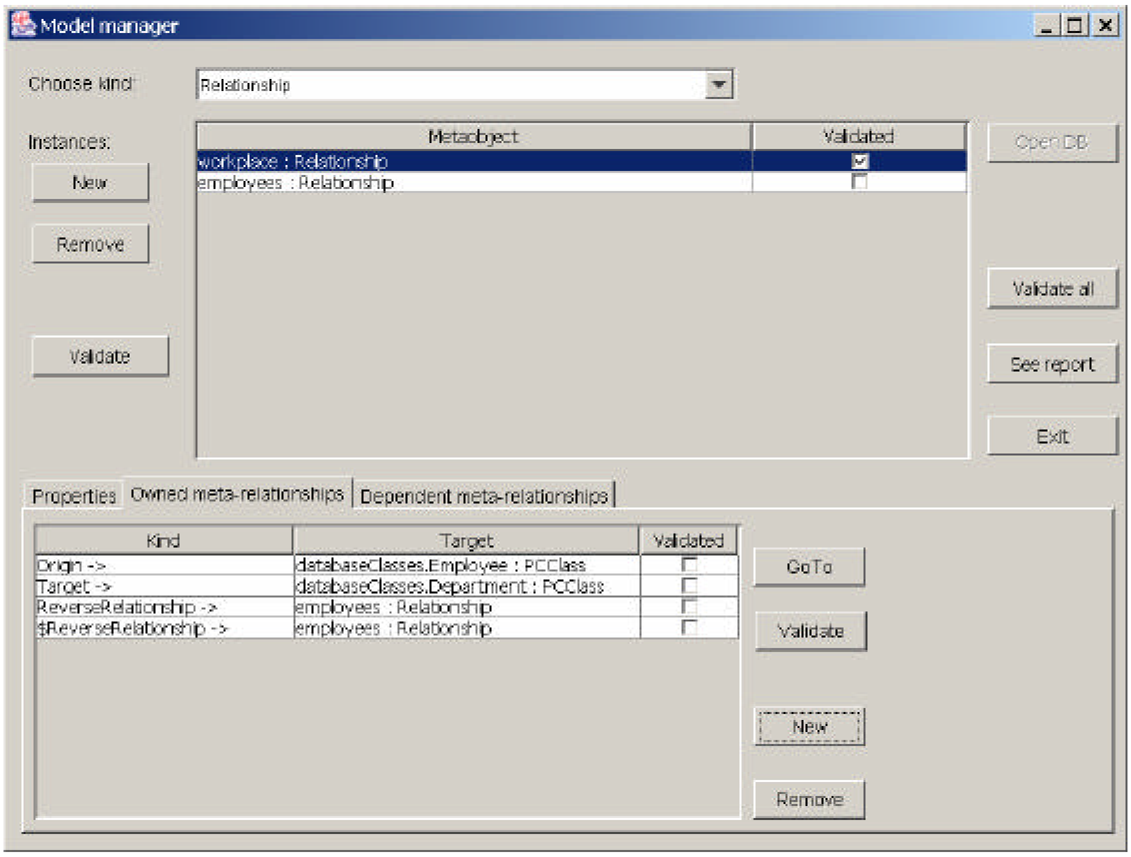
The central concept of this application is metaobject, thus all features (despite the whole model validation command) depend on the current selection of metaobject. The metaobjects are grouped according to their kind: selecting a kind name from the “Choose kind” combo list (see Fig. 26) results in showing all instances of that kind. Without selecting particular meta-object from the “Instances” table, it is possible to perform the following:

- Creation of a new meta-object of the kind determined by the current selection in the “Choose kind” combo;
- Validation of the whole model (that is, all metaobjects and all meta-relationships);
- Browsing the report from the above action.

Selecting a particular metaobject allows to:

- Perform an individual validation;
- Remove metaobject;

- Update metaobject's properties (that is – values of meta-attributes applicable to this kind of a metaobject – see Fig. 26).



Choose kind: Relationship

Metaobject	Validated
workplace : Relationship	☑
employees : Relationship	☐

Kind	Target	Validated
Origin ->	databaseClasses.Employee : PCClass	☐
Target ->	databaseClasses.Department : PCClass	☐
ReverseRelationship ->	employees : Relationship	☐
$ReverseRelationship ->	employees : Relationship	☐

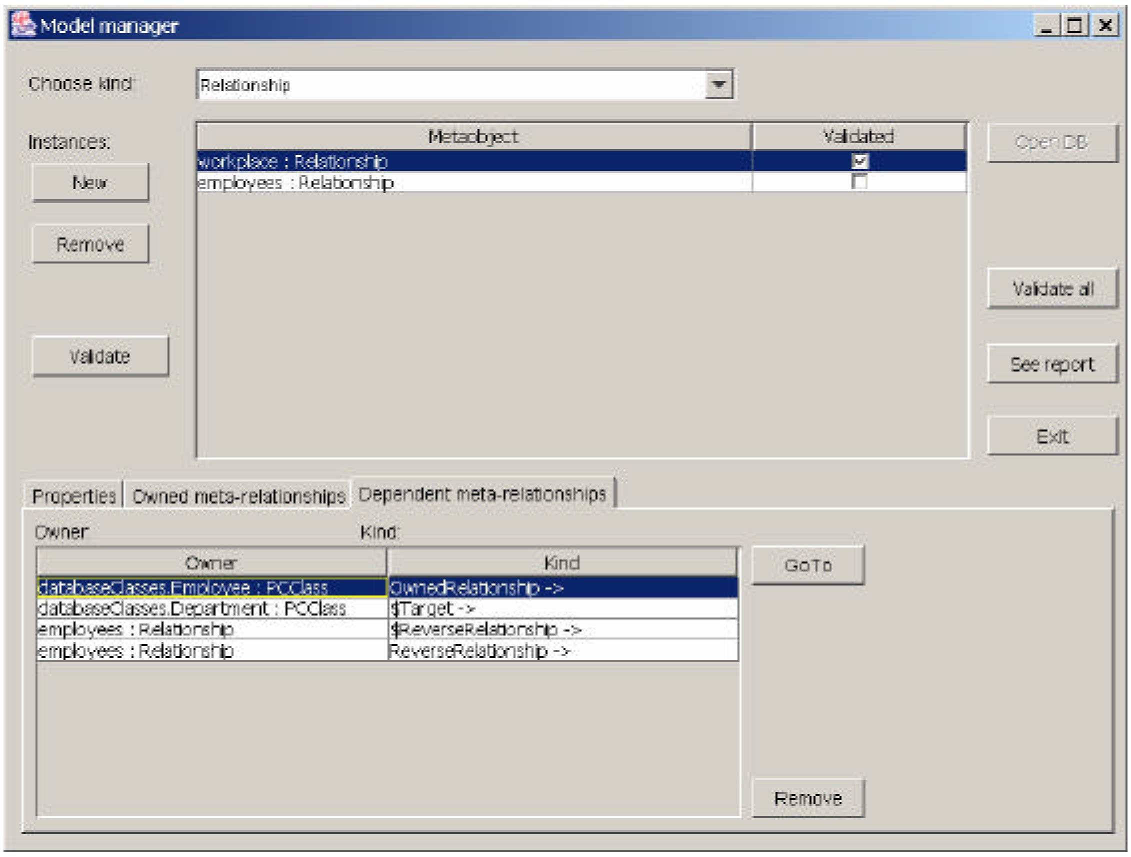
Fig. 27: Model Manager window – the "Owned meta-relationships" tag

Moreover, the 2-nd and 3-rd tag ("Owned meta-relationships" and "Dependent meta-relationships") allow to edit meta-relationships that are connected to the selected metaobject as appropriately its origin or its target (see Fig. 27 and Fig. 28).

Selecting particular owned meta-relationship allows to:

- Remove that meta-relationship;
- Validate it;
- Or to navigate to the target metaobject of this meta-relationship.

For convenience, the first and third of above options are available also in the "backward" direction (see Fig. 28), that is for the meta-relationships for which the selected meta-object is a target rather than an owner.



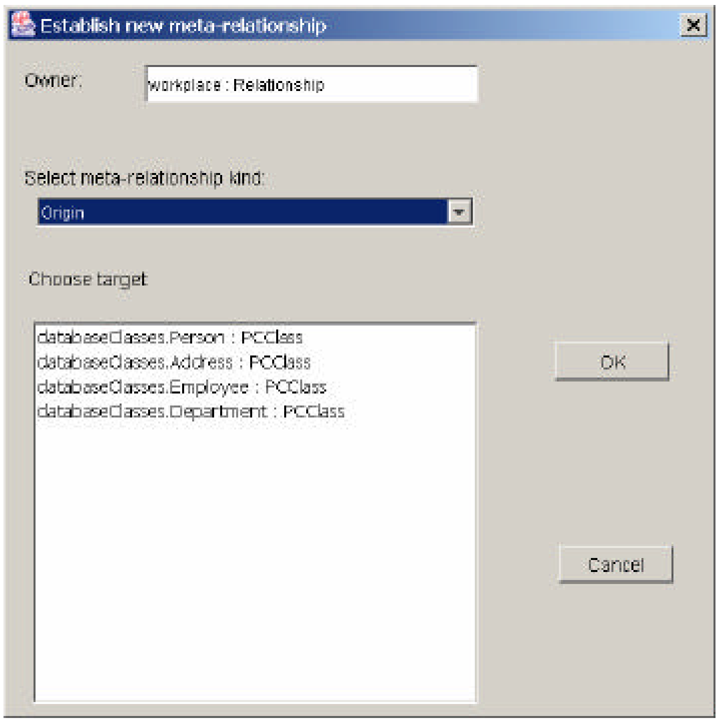
Choose kind: Relationship

Metaobject	Validated
workplace : Relationship	✓
employees : Relationship	☐

Owner: Kind:

Owner	Kind
databaseClasses.Employee : PCClass	OwnedRelationship ->
databaseClasses.Department : PCClass	$Target ->
employees : Relationship	$ReverseRelationship ->
employees : Relationship	ReverseRelationship ->

Fig. 28. Model Manager window – the "Dependent meta-relationships" tag



Owner: workplace : Relationship

Select meta-relationship kind: Origin

Choose target

databaseClasses.Person : PCClass
databaseClasses.Address : PCClass
databaseClasses.Employee : PCClass
databaseClasses.Department : PCClass

Fig. 29. Model Manager– the "Establish new meta-relationships" dialog

The “Owned meta-relationships” tag allows also for creating new meta-relationships. Thanks to the access to metamodel definition, the dialog dedicated to this option limits the possible selection to allowed combinations of metaobject kinds (Fig. 29). At the first stage it requires to select the meta-relationship name from among of meta-relationships the selected metaobject can own. Then, the “Choose target” list is being filled only with metaobjects, whose kinds are allowed as targets for the selected [owner-kind <–> meta-relationship name] combination.

Implementation classes of Model Manager

The implementation of Model Manager is based on the flattened metamodel structure, whose conceptual design was presented at the beginning of this chapter. Since it was assumed that both the meta-attribute and the meta-value are represented by just a single name, this structure was further reduced to only two project-specific classes: *MetaObject* and *MetaRelationship*. Fig. 30 shows practically complete interface of those classes, as well as their private attributes. Taking into account that majority of those methods are trivial and are included just to realize encapsulation, the structure may be classified as being very simple. In addition to those two classes the package *metadataRep.metamodel* also defined two interfaces: *MetaObjectValidator* and *MetaRelationshipValidator* used to connect metadata with validator classes defined by a metamodel designer.

The properties of metaobject are stored within a map (dictionary) structure, where the meta-attribute names are stored as keys and the strings representing meta-values form the value entries. The references to *MetaObjectValidator* and *MetaRelationshipValidator* are of class scope and in the current implementation lead to single *MetamodelDictionary* object. The responsibilities of *MetaObjectValidator* are the following:

- Validating the provided metaobject;
- Checking if a metaobject kind of the name provided exists;
- Providing a list of meta-attributes assigned to selected metaobject kind.

To realize the first of abovementioned tasks it is necessary for the validator to:

1. Check the kind of the provided metaobject.

2. Lookup the appropriate metaobject kind description.
3. Locate an object of class, which was designated as a validation code for that metaobject kind.
4. Forward the *validate(MetaObject)* message to that object.

Taking into account that the metaobject kind check and dispatching of the *validate* method are encapsulated within the *MetamodelDictionary* class, the validation invocation scenario is quite simple, as shown in Fig. 31.

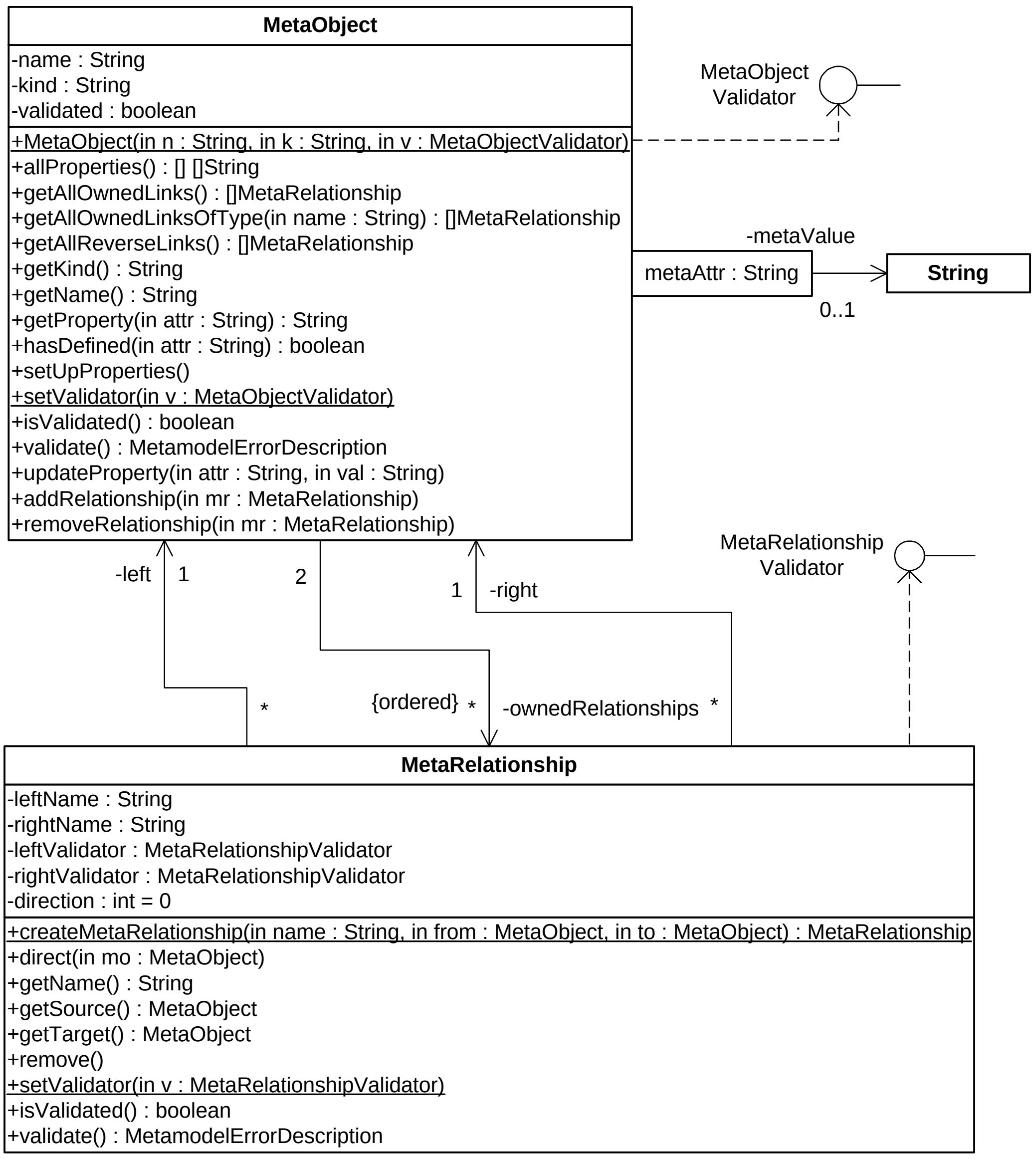

MetaObject
-name : String
-kind : String
-validated : boolean
+MetaObject(in n : String, in k : String, in v : MetaObjectValidator)
+allProperties() : [] []String
+getAllOwnedLinks() : []MetaRelationship
+getAllOwnedLinksOfType(in name : String) : []MetaRelationship
+getAllReverseLinks() : []MetaRelationship
+getKind() : String
+getName() : String
+getProperty(in attr : String) : String
+hasDefined(in attr : String) : boolean
+setUpProperties()
+setValidator(in v : MetaObjectValidator)
+isValidated() : boolean
+validate() : MetamodelErrorDescription
+updateProperty(in attr : String, in val : String)
+addRelationship(in mr : MetaRelationship)
+removeRelationship(in mr : MetaRelationship)
MetaObject Validator
-metaValue
metaAttr : String
String
0..1
MetaRelationship Validator
-left
1
2
1
-right
*
{ordered} *
-ownedRelationships *
MetaRelationship
-leftName : String
-rightName : String
-leftValidator : MetaRelationshipValidator
-rightValidator : MetaRelationshipValidator
-direction : int = 0
+createMetaRelationship(in name : String, in from : MetaObject, in to : MetaObject) : MetaRelationship
+direct(in mo : MetaObject)
+getName() : String
+getSource() : MetaObject
+getTarget() : MetaObject
+remove()
+setValidator(in v : MetaRelationshipValidator)
+isValidated() : boolean
+validate() : MetamodelErrorDescription

Fig. 30. The implementation of the flattened metamodel

The functionality of *MetaRelationshipValidator* is similar and consists of the following functions:

- Checking if a meta-relationship of the selected name is defined within metamodel;
- Validating the provided meta-relationship;
- Getting the name of relationship reverse to the one provided;
- Checking if the provided combination of meta-relationship name and a pair of metaobject kind names is allowed to instantiate.

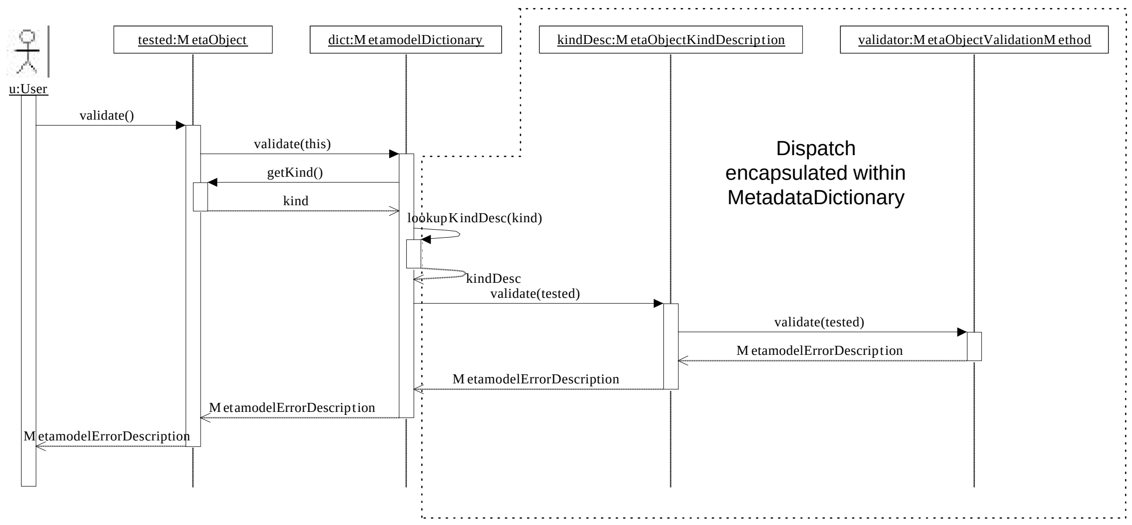

u:User
tested:MetaObject
dict:MetamodelDictionary
kindDesc:MetaObjectKindDescription
validator:MetaObjectValidationMethod
validate()
validate(this)
getKind()
kind
lookupKindDesc(kind)
kindDesc
validate(tested)
validate(tested)
MetamodelErrorDescription
MetamodelErrorDescription
MetamodelErrorDescription
MetamodelErrorDescription
Dispatch encapsulated within MetadataDictionary

Fig. 31. Invoking metaobject validation – a UML sequence diagram

As already mentioned, the dependencies between model and metamodel have to be bidirectional, and this remark seem to be of more general nature than just an implementational assumption made here. The diagram summarizing the dependencies of core concepts of this metamodel implementation is shown in Fig. 32. Note that the *metamodel* package is independent on any other package.

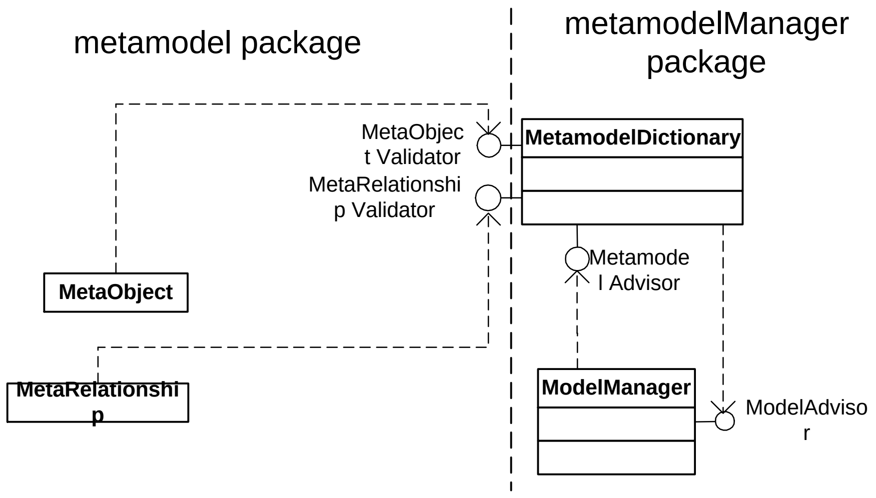


Fig. 32. Class diagram showing dependencies among core elements of the implemented metamodel

5.4 Database Analyzer

This part of implementation is a command-line application, designed to extract a schema of Objectivity/DB ODBMS into the flat metamodel structure. Since Java programmer's interface of this DBMS does not provide a direct access to its schema repository, the reflective capabilities of Java language were used to extract the metadata. The main drawback of this solution is that the persistence-capable class that is registered in the schema needs either to have at least one instance within the database or to be referenced by other persistence-capable class that has at least one instance. Otherwise such class would not be found by *Database Analyzer*. Taking into account the typical Objectivity for Java usage scenario (see chapter 2), this limitation does not seem to be severe.

With presence of the metamodel and model implementations described above with programming interface to access them, the realization of Database Analyzer is quite obvious and thus it will not be described here. Note also that since the metamodel is in this case fully determined by the Objectivity/DB architecture and Java object model, there will be no need to edit the metamodel definition stored within *MetamodelDictionary* (appropriate definitions are hard-coded in the initializing part of Database Analyzer). Similarly, the need to define metaobject and meta-relationship validators would be significantly reduced, since the metadata being extracted comes from compiled Java classes.[27]

Despite those facts the application is interesting, because of realizing a reasonably complete metamodel using the postulated flattened metadata structure. Moreover, the below description of Objectivity for Java metamodel shows, how the presented approach can address the contradictory requirements of expressiveness (desired for its descriptive role) and simplicity (implementational requirements). This is thanks to the fact that the expressive UML style of metamodel definition can be mapped into the flattened form in a very straightforward way, as described in the previous chapter.

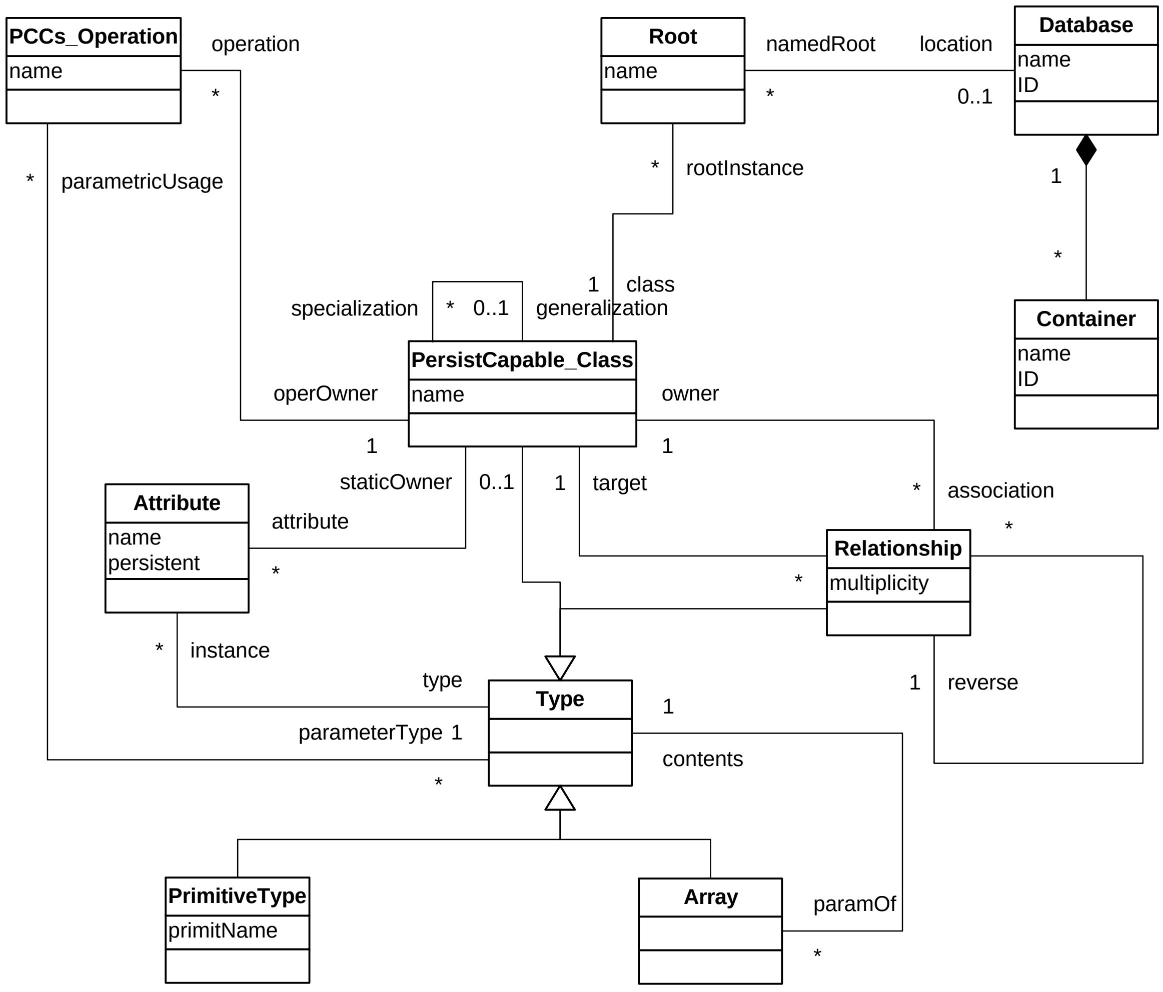


Fig. 33. Conceptual view of a simplified Objectivity for Java metamodel

[27] However, such checking elements can still be useful to enforce some more subtle properties of the design style.

The structure presented in Fig. 33 does not address all properties the selected DBMS supports. The following metadata kinds are considered in that simplified metamodel:

- All types used in definition of database classes (referred here as *Persistence-Capable Classes* – PCC), including built-in primitive types (in their Java binding name), arrays of different types (including multidimensional arrays) and finally PCCs (both predefined, like e.g. persistent collections, and application-defined ones).
- All attributes of each PCC, including transient ones.
- All bi-directional relationships between PCCs (more sophisticated, integrity-assuring substitute for plain object-referencing attributes).
- All operations defined for each PCC.
- All databases contained in the analyzed federated database and containers they consist of.
- Database root variables declaration: their names and references to PCCs they are instances of (both global roots – of federation scope, and local – defined for particular database).

The presented elements are the most important ones and all of them can be extracted using a combination of API operations provided in Java binding and (mainly) – the Java's reflection mechanism.

As already mentioned, the described module was necessary due to the fact that the Objectivity/DB schema is manipulated only internally by the DBMS. That is – metadata is not available to a programmer in a way analogous to regular data stored within database.[28] Taking into account various responsibilities of database metadata discussed in this work and especially the usefulness of custom or vendor-specific extensions to a metadata contents, such solution can be considered as a drawback. Despite positive aspects (e.g. simpler programmer's interface and easily achieved protection of schema against illegal updates), this can be considered as a factor limiting the DBMS

[28] Other options for extracting Objectivity schema contents were parsing "schema dump" in a form of a regular text file or resorting to the *Active Schema* feature available as an extension of the C++ binding [32].

functionality. Therefore, the most general postulate of this work should state that database metadata should be an extensible structure, directly available to programmer through the means analogous to those supporting regular data.

5.5 Dependency Discoverer

Having access to metadata structure (in our case the metadata extracted by *Database Analyzer* will be used), it is possible to extend it towards addressing of various additional features. As stated in the previous chapter, one of such tasks is the support for software configuration management in the area of database schema evolution, and this feature was implemented as an example of database metadata extension.

Metamodel extensions

According to the postulates from the previous chapter, the aim of this implementation is to collect information on broadly understood *backward dependencies* on database schema. This means the need to identify all procedural units accessing database together with specification of the kind of this access. Particular kinds of database access worth distinguishing are specific to a given data model, technology or even product. In case of Objectivity/DB Java binding, the following dependency kinds were identified:

- Operation call dependency (concerns calls of operations of PCCs from within any other methods).
- Side-effect dependencies (direct access to database objects' attributes). The read-only (RO) and read-write (RW) access kinds are distinguished.
- Root-lookup dependencies (method's attempt to bind a particular name of a root variable).
- Local (that is container- or database-scope) scans for database objects of particular class. This is a way of acquiring object references alternative to using root variables.[29]

[29] This operation accepts only a class name as a scan criterion. Another version of the *scan(..)* operation allows to specify simple predicates (in a form of string) to narrow the selection based on attribute values.

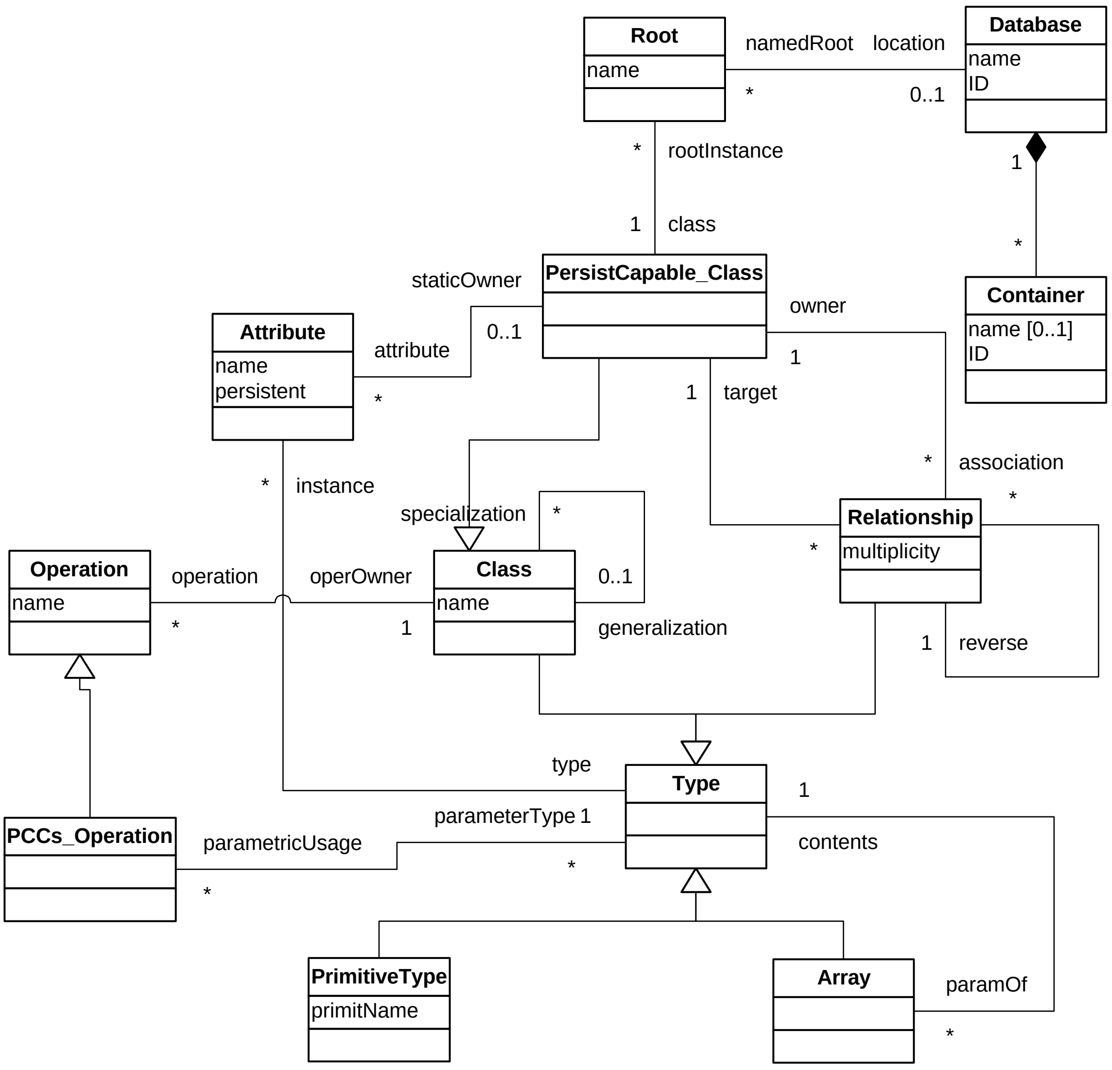


Fig. 34. Objectivity for Java metamodel prepared for defining external dependencies on DB schema

To support abovementioned information, the first step was a modification of a metamodel from Fig. 33 to incorporate elements identifying non-persistent application classes, together with those of their methods that access elements of database schema. Thus the appropriate metamodel concepts (that is, *Persistence-Capable Class (PCC)* and *PCCs Operation*) were generalized to cover external application elements (that is, *Class* and *Operation*) – see Fig. 34.

With these minor adjustment made, the support for schema dependency information requires only few additional associations (see Fig. 35):

Assuming that such selection could be the only case of usage of a given PCC's attribute by an external method, it seems valuable to consider extracting also the parameter names used in predicate.

- *Read-only* and *read-write* dependencies between operations and non-local attributes;
- Call dependency between operation and PCCs operation.
- Lookup dependency between operation and root variable.
- Scan dependency between operation and a storage object (database or container), concerning instances of particular class. This is conceptually a tertiary association, decomposed for implementation into a class (*Scan*) and three binary associations.

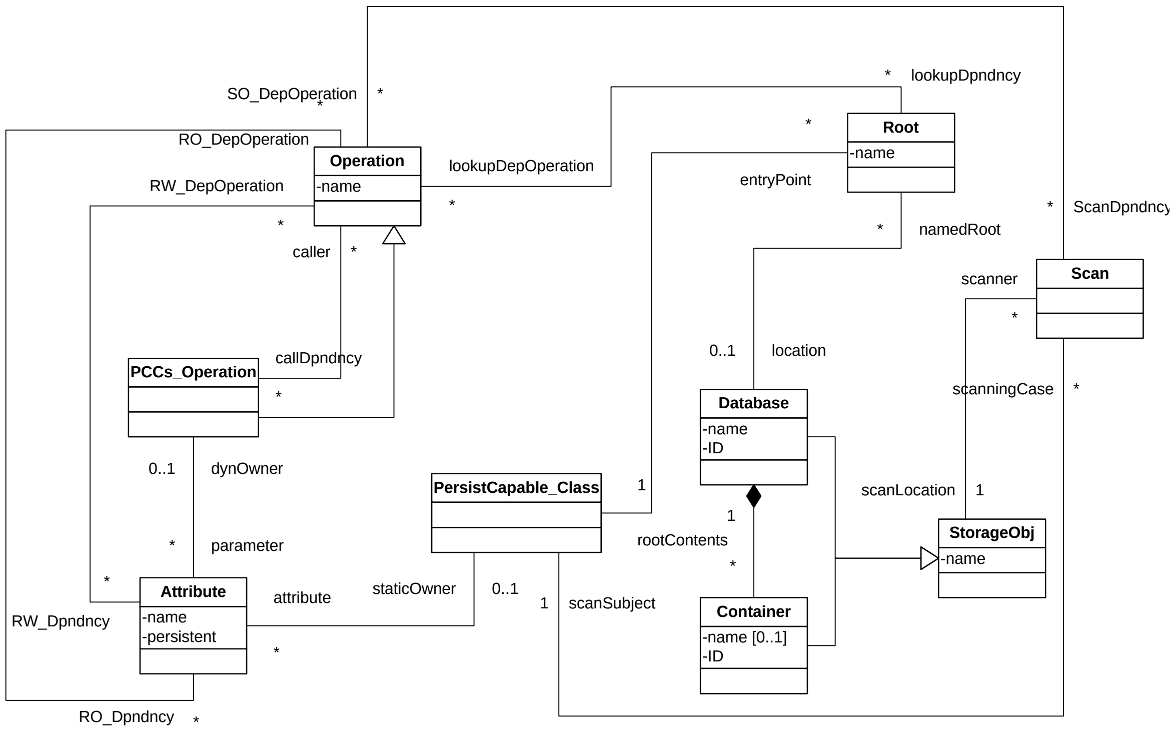


Fig. 35. Dependency-tracking elements of the Objectivity for Java metamodel

Collecting the dependency information

From among of possible ways of collecting the information on database dependencies, a variant of solution sketched in the previous chapter has been applied. That is, dependencies are detected and recorded during application testing and reflective capabilities are used to identify the caller. Due to architecture of Objectivity/DB, the mechanism registering dependencies is not a DBMS extension. Instead, it is (rather loosely) connected with applications' code, using aspect-oriented programming (AOP) Java extension (AspectJ [2]).

The problem was divided into two tasks: monitoring of currently executing methods and detecting any non local (that is – coming from another class) calls to database objects. The simplest solution of the former task, used in this implementation, is following:

- Defining *pointcuts* to intercept the start and the end of method's execution.
- Implementation of a stack of method description objects.
- Reflective extraction of method's signature at the moment of its invocation and pushing that data on the stack.
- Popping the method's description at the moment when its execution terminates.

This solution allowed to illustrate the idea using simple exemplary applications. However, it is not universal, taking into account a multiple-threaded execution model.[30] Nevertheless, these limitations can be easily removed and they do not affect the feasibility of presented solution in the Objectivity for Java environment.

The second task required introducing further *pointcuts*:

- Direct read / direct write of an attribute of database object.
- Call of an operation on a database object.
- Call of the *lookup(String name)* operation on a database or federation object (used to access a root object).
- Call of the *scan(String className, …)* operation.

Each occurrence of one of the above conditions triggers the following actions:

- Reading the top element of the executing methods' description stack;
- Lookup of that description within schema;
- If necessary – addition of the method description to the schema;
- Creating the appropriate association between method's description and schema element.

[30] This problem can be observed even in simple application, when the *toString()* operation (supported by any Java object) is invoked by GUI mechanism during form refreshing.

For performance reasons and conceptual clarity it is assumed that a complete schema description has been extracted by *Database Analyzer* prior to running the *Dependency Discoverer* feature.

5.6 Model Browser

This application provides a subset of the *Model Manager* functionality to allow for easier browsing of existing models. Evaluating this application can be interesting as a test for expressiveness or just the readability of the postulated flattened metamodel when manipulated by programmer.[31] This is because the browser preserves the inherent genericity of this approach, not trying to adjust the representation according to metaobject's kind or meta-relationship's name. Thus, all metaobjects are shown in a uniform way, as a composition of properties (metaobject – meta-value pairs and meta-relationships owned by a given metaobject) and allow navigation along their meta-relationships (Fig. 36).

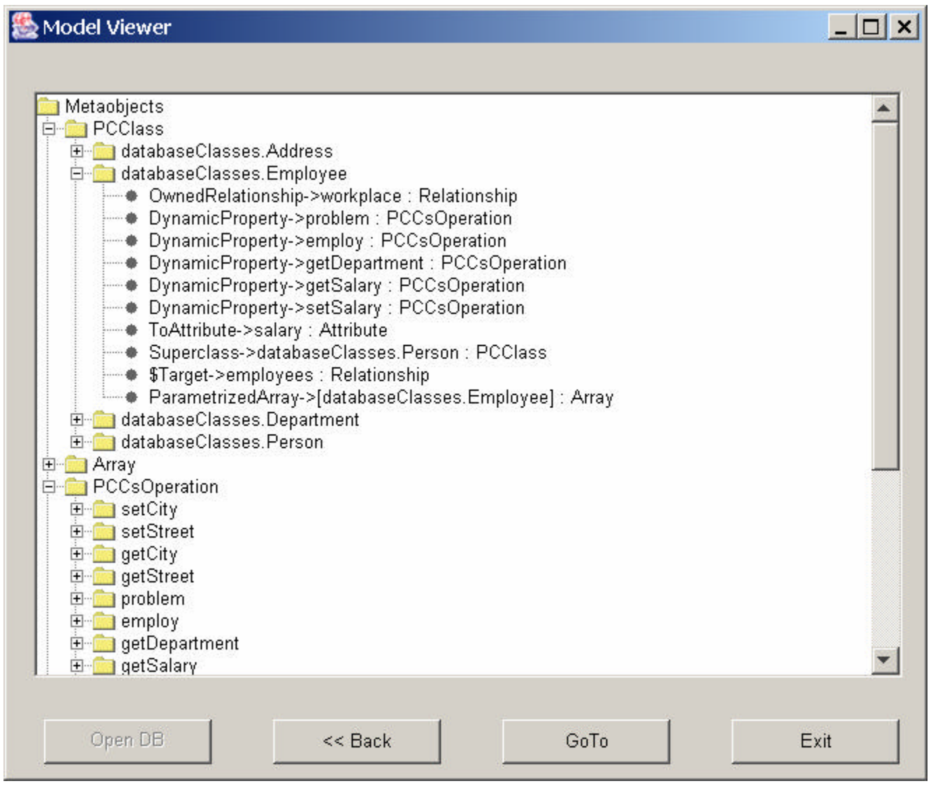

```
Metaobjects
⊟ PCClass
   ⊞ databaseClasses.Address
   ⊟ databaseClasses.Employee
      ● OwnedRelationship->workplace : Relationship
      ● DynamicProperty->problem : PCCsOperation
      ● DynamicProperty->employ : PCCsOperation
      ● DynamicProperty->getDepartment : PCCsOperation
      ● DynamicProperty->getSalary : PCCsOperation
      ● DynamicProperty->setSalary : PCCsOperation
      ● ToAttribute->salary : Attribute
      ● Superclass->databaseClasses.Person : PCClass
      ● $Target->employees : Relationship
      ● ParametrizedArray->[databaseClasses.Employee] : Array
   ⊞ databaseClasses.Department
   ⊞ databaseClasses.Person
⊞ Array
⊟ PCCsOperation
   ⊞ setCity
   ⊞ setStreet
   ⊞ getCity
   ⊞ getStreet
   ⊞ problem
   ⊞ employ
   ⊞ getDepartment
   ⊞ getSalary
```

Fig. 36. Model Browser window

[31] Only programmer or database administrator is supposed to deal directly with flattened metamodel structure. Thanks to straightforward mapping between rich UML-like metamodel and its flattened form the former can still be used for modeling and design as better suited for those tasks.

5.7 Implementation: encountered problems, remarks and conclusions

The implementation was realized using a pure, commercially available ODBMS that is – in an environment closely related to the one assumed by the ODMG standard, which was the starting point of this research.

The advantages of this environment are the close integration of DBMS functionality with programming language and the ability to directly utilize the core object-oriented constructs during implementation. On the other hand, the approach assuming the use of general-purpose programming language as a database data manipulation language, can be indeed considered controversial as being a relatively low-level solution. Despite the fact the ODMG standard defines an object query language OQL, its role, even within the standard definition, can be considered secondary. Moreover, a very limited interest of vendors in developing the OQL support into their products aggravates that trend.

Lack of (powerful enough) declarative means of selecting and modifying objects, made the implementation of the flattened metamodel less convenient. This is because that metamodel structure is strongly value-oriented; that is, the treatment of each element depends on its state (e.g. kind = "Interface") rather than on its type. Of course, this also leads to using generic means of manipulating the metadata, which is also rather against the ODMG metamodel philosophy.

Another assumption of the Objectivity/DB DBMS is an extremely decentralized distribution model. One of assumptions connected with this approach (and represented by the ODMG standard as well), was moving all the processing (together with a whole definition of object's behavior) onto the application side. Although this may result in a simpler architecture, lack of methods stored within a database may become in some cases very unnatural. This is because some operations serve as means to maintain consistency rules inherent to a given model definition rather that as an adjustable external interface. Consider for example situation where the same objects stored in a persistent collection are accessed by two applications written in different programming languages (e.g. Java and C++). Both applications need to provide their own code implementing all used operations of database objects. This is not only redundant but also inconsistency-prone solution. E.g. the comparator operations needed to maintain order in sorted collections may corrupt data if their implementations in both applications are not compliant.

Practical usage of the flattened metadata structure in *Database Analyzer* and *Dependency Discoverer* features proves its usability. However, it is clear that implementation could be realized in a more convenient way if a query language to manipulate metadata were available.

The realization of the dependency-tracking mechanism shows the importance of the extensibility concerning both database mechanisms as well as an externally accessible and extensible database metadata structure. The features serving to identify procedural unit accessing database proved to be highly dependent on a programming language. Structural and behavioral reflection also became necessary. This also exemplifies the problems connected with multi-language direct access to an ODBMS.

6 Conclusions and future work

The goal of this work was the investigation of the required features of a metamodel for ODBMS in the context of existing standardization efforts, particularly, the ODMG standard. The analysis showed a number of issues with the existing ODMG metamodel definition, which need to be solved, as well as some new requirements, not considered in the original standard at all.

6.1 ODBMS metamodel roles and suggested solutions

Probably the most broadly known object-oriented metamodel is the metamodel definition from the UML standard. This specification exemplifies the descriptive role of a metamodel: it is needed to properly and precisely understand the meaning of a given language's or tool's constructs, their interrelations, constraints and intended usage.

Despite its informal and meta-circular[32] style, the UML metamodel is quite successful in providing such description. Thus it seems to be acceptable to define the ODBMS metamodel in a very similar style.[33] This would be sufficient especially considering, that in case of a DBMS the metamodel concepts are related to a storage model and to a query language semantics, which make them more precise.

Another role of a DBMS metamodel is related to the fact, that it constitutes a foundation to implement a database schema repository, necessary for various DBMS operations. Such repository stores also physical data structure information, privacy and security information and other data that may be needed for optimization. All this information needs to be incorporated in a way that guarantees a good performance, since such metadata is expected to be accessed very frequently. Those kinds of metadata, that need to be explicitly used by a programmer, should be accessible through a simple and efficient interface. Both these requirements speak in favor of employing a metadata structure much simpler than the one assumed by the ODMG.

[32] As already mentioned, the UML definition is recursive. The term "meta-circular" means that the language definition is provided using a specially chosen subset of its own basic elements, which is called the *UML core*.

[33] Due to the popularity of the UML and MOF standards, the subset of UML could be a good choice as the basic mean to describe an ODBMS metamodel proposal.

The most obvious example of the need of making the database metadata externally available is the generic programming through reflection, which proved to be a very useful technique. The comparison with specifications of CORBA Interface Repository or Dynamic SQL [9] shows, that the current ODMG standard lacks some elements necessary to guarantee true portability of generic database applications.

Moreover, there are various requirements indicating the need of further extensions to the metamodel. One important source of such changes would be future additions of new conceptual modeling notions. An example would be the dynamic object role concept, whose influence on the considered database metamodel has been discussed here. Other, even more certain source of the future metamodel extensions would be the incorporation of important DBMS features, not considered in the current ODMG proposal. As suggested e.g. in [46], the most necessary subject of standardization would be a view mechanism and stored behavior (in the form of database procedures or perhaps also a kind of active rule mechanism).

Apart from abovementioned conventional database responsibilities, new metadata-related features should be considered. One example is support for SCM, assuming storage of database-related software dependencies within the schema repository. The integration and interoperability of independently developed systems requires much more meta-information that the data structure description stored traditionally in the schema. Thus, another important feature would be RDF-style descriptions to express database's ontology.

As can be seen from the above summary, the database metamodel has to deal with a very large number of various kinds of metadata, being at the same time well prepared for efficient implementation in a form of a schema repository, as well as for future evolution and custom extensions. For those reasons, a radically simplified ("flattened") metamodel to use for the implementation and manipulation of a schema repository has been proposed. This "lightweight" solution provides the level of flexibility and extensibility comparable with the traditional four-level metalevel architecture. On the other hand, to support expressiveness, the conceptual view of the proposed metamodel has been described in the UML style, and the simple rules of mapping it to the flattened form were provided.

Since the flattened metamodel structure differs significantly from the existing proposals in this area, the prototype implementation of a schema repository based on it has been realized. Although less expressive when directly drawn as a UML diagram, the flattened structure proved to be convenient for metadata manipulation. To provide a practical example, the flattened metamodel has been used to express the Objectivity/DB ODBMS schema elements. This served to provide a working illustration of another idea presented in this work that is, extending the DBMS schema to support the SCM through storing software dependency information.

6.2 Future work

Although this work attempts to provide a complete overview of the roles an ODBMS metamodel has to fulfill, many of suggested improvements and solutions require further research in order to provide a more specific proposal. The most important areas requiring a detailed solution include:

- **The view mechanism**, allowing, as far as possible, to treat the virtual objects as if they was regular objects. Thus the view definition need to be described as a fully-fledged sub-schema, including virtual objects' classes, their properties etc.
- **Behavioral elements**, including stored database procedures and active rules.
- **Security mechanisms**, standardized as a DBMS feature rather than some externally-defined facility.
- **Database distribution and interoperability**, which requires to explicitly deal with e.g. the *site* concept within a metamodel, in order to unambiguously identify and partition schema definitions and data sources.

The realization of postulated new features of database metamodel requires further investigation, e.g. in order to answer the following questions:

- What level of customizability and support for the *separation of concerns* principle needs to be provided to a database designer?
- How to standardize the vocabulary for the DB schema-based resource description system? To what extent it can be based on or unified with the RDF specification?

- What is the optimum way to collect the software dependency information concerning database, in order to store it within the schema?

Any proposal concerning database metamodel should respect as far as possible the established standards, their notions and existing vocabulary. Thus it was assumed here that the conceptual view of proposed metamodel features should be described using UML/MOF and their mapping into the flattened form (used for schema implementation and metadata manipulation) should be provided. The same style of description is intended to be used for future more detailed solutions of above outlined problems.

Although the metadata repository implemented and described in this work allowed to test some of the presented ideas, many details discussed here can be investigated only in presence of a fully-fledged ODBMS prototype. This concerns e.g. using a query language for metadata manipulation. Assuming the flattened metamodel, this seems to be very promising for simplifying a database metadata management, which is the main goal of this research.

On the other hand, the proposed flattened metamodel can be, thanks to its inherent flexibility, very well prepared for experimenting with different detailed solutions during the development of a prototype ODBMS.

7 Bibliography

[1] S. R. Alpert. Primitive Types Considered Harmful. Java Report, November, 1998 (Vol. 3, No 11).

[2] AspectJ project website: http://aspectj.org

[3] C.Bachman, M.Daya. The Role Concept in Data Models. Proc. of the 3rd VLDB Conf., Tokyo, pp.464-476 1977.

[4] J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou. Data Model Issues for Object-Oriented Applications. ACM Transactions for Office Information Systems, April 1987.

[5] G. Booch, I. Jacobson, J. Rumbaugh. The Unified Modeling Language User Guide. Addison-Wesley 1998.

[6] S. Brandt, R. W. Schmidt. The Design of a Metalevel Architecture for the Beta Language. In Advances in Object-Oriented Metalevel Architectures and Reflection, CRC Press 1996.

[7] A. B. Chaudhri, R. Zicari. Succeeding with Object Databases. Wiley 2001.

[8] K. T. Claypool, J. Jin, E. A. Rundensteiner. OQL SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework. In Centre for Advanced Studies Conference, 1998, 108-122.

[9] C. J. Date, H. Darwen. A Guide to the SQL Standard. Addison-Wesley 1997.

[10] DB4o – Database for Objects website: http://www.db4o.com

[11] Dublin Core Metadata Element Set, Version 1.1. http://dublincore.org/documents/

[12] E. W. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976.

[13] F. Ferrandina, S.-E. Lautemann. An Integrated Approach to Schema Evolution for Object Databases. OOIS 1996, 280-294.

[14] I. R. Forman, S. H. Danforth. Putting Metaclasses to Work. Addison-Wesley 1999.

[15] M. Fowler. Dealing with Roles. http://www.martinfowler.com/ (an update to: Analysis Patterns: Reusable Object Models. Addison-Wesley 1996).

[16] R. Geisler, M. Klar and S. Mann. Precise UML Semantics Through Formal Metamodeling. Proceedings of the OOPSLA’98 Workshop on Formalizing UML, 1998.

[17] I. A. Goralwalla, D. Szafron, M. T. Özsu, R. J. Peters. A Temporal Approach to Managing Schema Evolution in Object Database Systems. DKE 28(1), 1998, 73-105.

[18] G. Gottlob, M. Schrefl, B. Rock. Extending Object-Oriented Systems With Roles. ACM Transactions on Information Systems, pp. 268-296, 1996.

[19] P. Harmon. The Reinvention of the Omg. In: Distributed Enterprise Architecture, vol. 5 no. 1 / 2002.

[20] P. Harmon. The OMG's Model Driven Architecture. In: Component Development Strategies, vol. XII No. 1, January 2002 (available from http://www.omg.org).

[21] ICONS (*Intelligent CONtent Management System*) website: http://www.icons.rodan.pl/

[22] IEEE Guide to Software Configuration Management, ANSI/IEEE Std 1042-1987.

[23] IEEE Standard Glossary of Software Engineering Technology, ANSI/IEEE Std 610.12-1990.

[24] IEEE Standard for Software Configuration Management Plans. ANSI/IEEE Std 828 – 1990.

[25] ISO/IEC 12207. Information Technology - Software LifeCycle Processes. ISO/IEC Copyright Office, Geneva, Switzerland, 1995.

[26] SO/IEC TR 15846:1998 Information technology - Software life cycle processes - Configuration Management.

[27] Java Data Object (JDO) Specification. Version 1.0. Sun Microsystems 2002.

[28] A. Jodlowski, P. Habela, J. Plodzien, K. Subieta: Objects and Roles in the Stack-Based Approach. DEXA 2002: 514-523.

[29] D. Kambur, M. Roantree: Using Stored Behaviour in Object-Oriented Databases. EFIS 2001: 61-69.

[30] G. Kiczales, J. Lamping, A. Mendhekar, Ch. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin: Aspect-Oriented Programming. ECOOP 1997: 220-242.

[31] W. Kim. Observations on the ODMG-93 Proposal for an Object-Oriented Database Language. ACM SIGMOD Record, 23(1), 1994, 4-9.

[32] Objectivity/C++ Active Schema, Release 7.0. Objectivity, Inc. 2001.

[33] Objectivity for Java Programmer's Guide, Release 7.0. Objectivity, Inc. 2001.

[34] R. G. G. Cattell, D. K. Barry: The Object Data Standard: ODMG 3.0. Morgan Kaufmann 2000.

[35] Object Management Group: The Common Object Request Broker: Architecture and Specification. Version 3.0, July 2002 [http://www.omg.org].

[36] Object Management Group: Common Warehouse Metamodel (CWM) Specification. Version 1.0, October 2001 [http://www.omg.org].

[37] Object Management Group: UML Profile for Enterprise Distributed Object Computing Specification. Version 1.0, October 2001 [http://www.omg.org].

[38] Object Management Group: Model Driven Architecture (MDA). July 2001 (draft) [http://www.omg.org].

[39] Object Management Group: Meta Object Facility (MOF) Specification. February 2002 [http://www.omg.org].

[40] Object Management Group: Security Service Specification. Version 1.8, March 2002 [http://www.omg.org].

[41] Object Management Group: Unified Modeling Language (UML) Specification. Version 1.4, September 2001 [http://www.omg.org].

[42] Object Management Group: OMG XML Metadata Interchange (XMI) Specification. Version 1.2, January 2002 [http://www.omg.org].

[43] R. J. Peters, M. T. Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. TODS 22(1), 1997 75-114.

[44] J. Plodzien, K. Subieta: Applying Low-Level Query Optimization Techniques by Rewriting. DEXA 2001: 867-876.

[45] Y.-G. Ra, E. A. Rundensteiner. A Transparent Object-Oriented Schema Change Approach Using View Evolution. ICDE, 1995, 165-172.

[46] M. Roantree, J. Murphy and W. Hasselbring. The OASIS Multidatabase Prototype. ACM Sigmod Record, 28:1, March 1999.

[47] M. Roantree, K. Subieta: Generic Applications for Object-Oriented Databases. OOIS 2002: 53-59.

[48] D. Slama, J. Garbis, P. Russell. Enterprise CORBA. Prentice Hall PTR, 1999.

[49] F. Steimann: A Radical Revision of UML's Role Concept. UML 2000: 194-209.

[50] F. Steimann: On the representation of roles in object-oriented and conceptual modelling. DKE 35(1): 83-106 (2000).

[51] H. Su, K. T. Claypool, E. A. Rundensteiner. Extending the Object Query Language for Transparent Metadata Access. Database Schema Evolution and Meta-Modeling, 9th Intl. Workshop on Foundations of Models and Languages for Data and Objects, 2000, Springer LNCS 2065, 2001 182-201.

[52] K. Subieta, C. Beeri, F. Matthes, J. W. Schmidt: A Stack-Based Approach to Query Languages. East/West Database Workshop 1994: 159-180.

[53] K. Subieta, M. Missala: Semantics of Query Languages for the Entity-Relationship Model. ER 1986: 197-216.

[54] K. Subieta: Object-Oriented Standards: Can ODMG OQL be Extented to a Programming Language? CODAS 1996: 459-468.

[55] K. Subieta. Object-Oriented Standards. Can ODMG OQL Be Extended to a Programming Language? Proc. of International Symposium on Cooperative Database Systems, Kyoto, Japan, December 1996. In: Cooperative Databases and Applications, World Scientific, 1997, 459-468.

[56] K. Subieta. Mapping Heterogenous Ontologies through Object Views. Proc. of 3-rd Workshop Engineering Federated Information Systems (EFIS 2000), Dublin, IOS Press, June 2000, 1-10.

[57] K. Subieta, M. Missala, K. Anacki. The LOQIS System, Description and Programmer Manual. Institute of Computer Science Polish Academy of Sciences Report 695, 1990.

[58] M. Tresch, M. H. Scholl. Meta Object Management and its Application to Database Evolution. ER 1992, 299-321.

[59] R. K. Wong, H. L. Chau, F. H. Lochovsky. A Data Model and Semantics of Objects with Dynamic Roles. Proc. of Intl. Conf. on Data Engineering, 1997.

[60] The World Wide Web Consortium. Resource Description Framework (RDF) Model and Syntax Specification. February 1999 [http://w3.org].

[61] The World Wide Web Consortium. RDF Vocabulary Description Language 1.0: RDF Schema. November 2002 (working draft) [http://w3.org].

[62] The World Wide Web Consortium. Extensible Markup Language (XML) 1.0. October 2000 [http://w3.org].

[63] The World Wide Web Consortium. XML Schema Part 0: Primer. May 2001 [http://w3.org].

8 Appendices

A. Abbreviations

AOP	Aspect-Oriented Programming
CORBA	Common Object-Request Broker Architecture
CVS	Concurrent Versioning System
CWM	Common Warehouse Metamodel
DC	Dublin Core [Metadata Element Set]
DII	Dynamic Invocation Interface
DBMS	Database Management System
DOI	Digital Object Identifier
DSI	Dynamic Skeleton Interface
DTD	Document Type Definition
EJB	Enterprise Java Bean
GUI	Graphical User Interface
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IR	Interface Repository
MDA	Model Driven Architecture
MLI	Meta Level Interception
MOF	Meta Object Facility
OASIS	ODMG Architectures for the Specification of Interoperable Systems
OCL	Object Constraint Language
ODBMS	Object-Oriented Database Management System
ODL	Object Definition Language
ODMG	Object Data Management Group
OLAP	On-Line Analytical Processing
OMG	Object Management Group
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering
OQL	Object Query Language
PCC	Persistence-Capable Class
SCI	Software Configuration Item
SCM	Software Configuration Management
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

B. Objectivity for Java simplified metamodel elements

This appendix presents the complete set of metaobject kinds and meta-relationship names used to express the necessary elements of the Objectivity/DB metamodel (as expressed in Fig. 34 and Fig. 35) in the flattened form. Since the well-formedness of the extracted schema is guaranteed by the mechanisms of the Objectivity/DB DBMS, there was no need to implement the appropriate constraints to verify it. The presented data is included within the *Database Analyzer* code and is registered into the database before starting the schema analysis of the provided database.

First the flat metamodel definition for Objectivity/DB is presented from the point of view of particular metaobject kinds. An asterix (*) preceding a meta-relationship name or a metaobject kind indicates, it is an extension introduced to store the dependency information rather than a part of the original Objectivity/DB schema. The further part of this appendix enumerates the defined meta-relationship names together with metaobject kind combinations allowed to be connected by them.

External Operation

Kind	*External Operation
Meta-attributes	-
Owned meta-relationships kinds	Call Dependency -> PCs Operation Operation Owner -> *External Class To Parameter -> Parameter Return Type -> PC Class Return Type -> External Class Return Type -> Array Return Type -> Primitive Type *RO Dependency -> Attribute *RW Dependency -> Attribute *Scan Dependency -> Scan *Lookup Dependency -> Root *Call Dependency -> PCs Operation
Constraints	Exactly one Operation Owner

PCCs Operation

Kind	PCCs Operation
Meta-attributes	-
Owned meta-relationships kinds	Call Dependency -> PCCs Operation Operation Owner -> PC Class To Parameter -> Parameter Return Type -> PC Class Return Type -> *External Class Return Type -> Array Return Type -> Primitive Type *Caller Operation -> *External Operation *Caller Operation -> PCCs Operation *RO Dependency -> Attribute *RW Dependency -> Attribute *Scan Dependency -> Scan *Lookup Dependency -> Root *Call Dependency -> PCs Operation
Constraints	Exactly one Operation Owner

External Class

Kind	*External Class
Meta-attributes	-
Owned meta-relationships kinds	Dynamic Property -> *External Operation Instance -> Parameter Instance -> Attribute
Constraints	-

PC Class

Kind	PC Class
Meta-attributes	-
Owned meta-relationships kinds	Dynamic Property -> PCCs Operation Owned Relationship -> Relationship To Attribute -> Attribute Entry Point -> Root Instance -> Attribute Instance -> Parameter Superclass -> PC Class Subclass -> PC Class *Scanning Case -> Scan
Constraints	-

Scan

Kind	*Scan
Meta-attributes	Formula
Owned meta-relationships kinds	*Scan Location -> Database *Scan Location -> Container *Scan Subject -> PC Class *Scan Dependent Operation -> External Operation *Scan Dependent Operation -> PCs Operation
Constraints	-

Attribute

Kind	Attribute
Meta-attributes	-
Owned meta-relationships kinds	Type -> PC Class Type -> Primitive Type Type -> Array Type -> *External Class Attribute Owner -> PC Class *RO Dependent Operation -> External Operation *RO Dependent Operation -> PCs Operation *RW Dependent Operation -> External Operation *RW Dependent Operation -> PCs Operation
Constraints	-

Parameter

Kind	Parameter
Meta-attributes	Position
Owned meta-relationships kinds	Type -> PC Class Type -> *External Class Type -> Primitive Type Type -> Array Parameter Owner -> PCs Operation Parameter Owner -> *External Operation
Constraints	-

Relationship

Kind	Relationship
Meta-attributes	Multiplicity
Owned meta-relationships kinds	Origin -> PC Class Target -> PC Class Reverse Relationship -> Relationship
Constraints	Exactly one reverse rel.: symmetrical, non recursive

Primitive Type

Kind	Primitive Type
Meta-attributes	-
Owned meta-relationships kinds	Instance -> Attribute Instance -> Parameter Parametrized Array -> Array
Constraints	-

Array

Kind	Array
Meta-attributes	-
Owned meta-relationships kinds	Contents -> Primitive Type Contents -> Array Contents -> *External Class Contents -> PC Class
Constraints	-

Database

Kind	Database
Meta-attributes	ID
Owned meta-relationships kinds	Component -> Container Maintained Root Name -> Root *Scanner -> *Scan
Constraints	-

Container

Kind	Container
Meta-attributes	ID
Owned meta-relationships kinds	Composition -> Database *Scanner -> *Scan
Constraints	-

Root

Kind	Root
Meta-attributes	-
Owned meta-relationships kinds	Location -> Database Contents Type -> PC Class *Lookup Dependent Operation -> *External Operation *Lookup Dependent Operation -> *External Operation
Constraints	Maximum one location. Exactly one Type (is obligatory).

Meta-Relationships

Call Dependency	(External Operation	->	PCCs Operation)
	(PCCs Operation	->	PCCs Operation)
+REVERSE			
Caller	(PCCs Operation	->	PCCs Operation)
	(PCCs Operation	->	External Operation)
Operation Owner	(External Operation	->	External Class)
	(PCCs Operation	->	PCCs Class)
+REVERSE			
Dynamic Property	(External Class	->	External Operation)
	(PCCs Class	->	PCCs Operation)
Owned Relationship	(PC Class	->	Relationship)
+REVERSE			
Origin	(Relationship	->	PC Class)
Target	(Relationship	->	PC Class)
(NO REVERSE)			
Reverse Relationship	(Relationship	->	Relationship)
(NO REVERSE)			
To Attribute	(PC Class	->	Attribute)
+REVERSE			
Attribute Owner	(Attribute	->	PC Class)
To Parameter	(PCs Operation	->	Parameter)
	(External Operation)	->	Parameter)
+REVERSE			
Parameter Owner	(Parameter	->	PCs Operation)
	(Parameter	->	External Operation)

Type	(Attribute	->	PC Class)
	(Attribute	->	Array)
	(Attribute	->	Primitive Type)
	(Attribute	->	External Class)
	(Parameter	->	PC Class)
	(Parameter	->	External Class)
	(Parameter	->	Array)
	(Parameter	->	Primitive Type)
+REVERSE			
Instance	(PC Class	->	Attribute)
	(Array	->	Attribute)
	(Primitive type	->	Attribute)
	(External Class	->	Attribute)
	(PC Class	->	Parameter)
	(External Class	->	Parameter)
	(Array	->	Parameter)
	(Primitive type	->	Parameter)
Return Type	(External Operation	->	PC Class)
	(External Operation	->	Array)
	(External Operation	->	Primitive Type)
	(External Operation	->	External Class)
	(PCCs Operation	->	External Class)
	(PCCs Operation	->	PC Class)
	(PCCs Operation	->	Array)
	(PCCs Operation	->	Primitive Type)
+REVERSE			
Returning Operation	(PC Class	->	External Operation)
	(External Class	->	External Operation)
	(Array	->	External Operation)
	(Primitive type	->	External Operation)
	(PC Class	->	PCCs Operation)
	(External Class	->	PCCs Operation)
	(Array	->	PCCs Operation)
	(Primitive type	->	PCCs Operation)

Contents	(Array	->	Primitive Type)
	(Array	->	Array)
	(Array	->	External Class)
	(Array	->	PC Class)
(+REVERSE)			
Parametrized Array	(Primitive Type	->	Array)
	(Array	->	Array)
	(External Class	->	Array)
	(PC Class	->	Array)
Composition	(Container	->	Database)
(+REVERSE)			
Component	(Database	->	Container)
Maintained Root Name	(Database	->	Root)
(+REVERSE)			
Location	(Root	->	Database)
Entry Point	(PC Class	->	Root)
(+REVERSE)			
Contents Type	(Root	->	PC Class)
***RO Dependency**	(*External Operation	->	Attribute)
	(PCs Operation	->	Attribute)
(+REVERSE)			
***RO Dependent Operation**	(Attribute	->	External Operation)
	(Attribute	->	PCs Operation)
***RW Dependency**	(*External Operation	->	Attribute)
	(PCs Operation	->	Attribute)
(+REVERSE)			
***RW Dependent Operation**	(Attribute	->	External Operation)
	(Attribute	->	PCs Operation)

***Lookup Dependency**	(External Operation	->	Root)
	(PCs Operation	->	Root)
(+REVERSE)			
***Lookup Dependent Operation**	(Root	->	External Operation)
	(Root	->	PCs Operation)
***Scan Dependency**	(External Operation	->	Scan)
	(PCs Operation	->	Scan)
(+REVERSE)			
***Scan Dependent Operation**	(Scan	->	External Operation)
	(Scan	->	PCs Operation)
***Scan Location**	(Scan	->	Database)
	(Scan	->	Container)
(+REVERSE)			
***Scanner**	(Database	->	Scan)
	(Container	->	Scan)

C. Test cases for Database Analyzer and Dependency Discoverer applications

This appendix describes two mini-applications that were used to test the functioning of dependency-tracking mechanism. The aim was to provide a minimum functionality needed to test the discovering of all distinguished kinds of dependencies concerning database. Because of prototype character of all the software created in connection with this work, it has not been tested very thoroughly. However, it was possible to observe, that all kinds of dependencies intended to be tracked, can be easily discovered without the need of changing original applications thanks to the use of the AspectJ [2] language AOP capabilities.

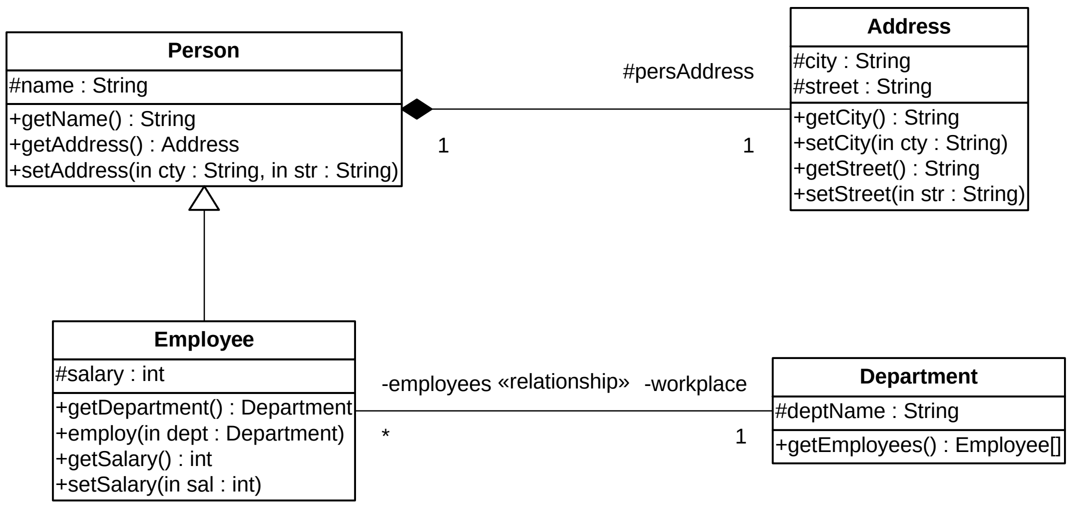


Fig. 37. The persistence-capable classes defined for test applications "Address book" and "Department-Employee" (a UML class diagram)

The persistence-capable classes defined for those applications and contained in a separate Java package are presented using UML class diagram in Fig. 37. They constitute a part of database schema information, the test applications depend on. All other classes, that is those providing the GUI, as well as the main application classes, are considered to be external to the database schema. The dependencies of interest are those coming from either an external class or schema-defining class, with exception of the calls local to a given class (e.g. between the *getName()* method in class *Person* and its *name* attribute).

Below the functionality of both applications is presented and annotated to describe the mechanisms used intentionally to make all interesting dependency kinds occur.

“Address Book” application

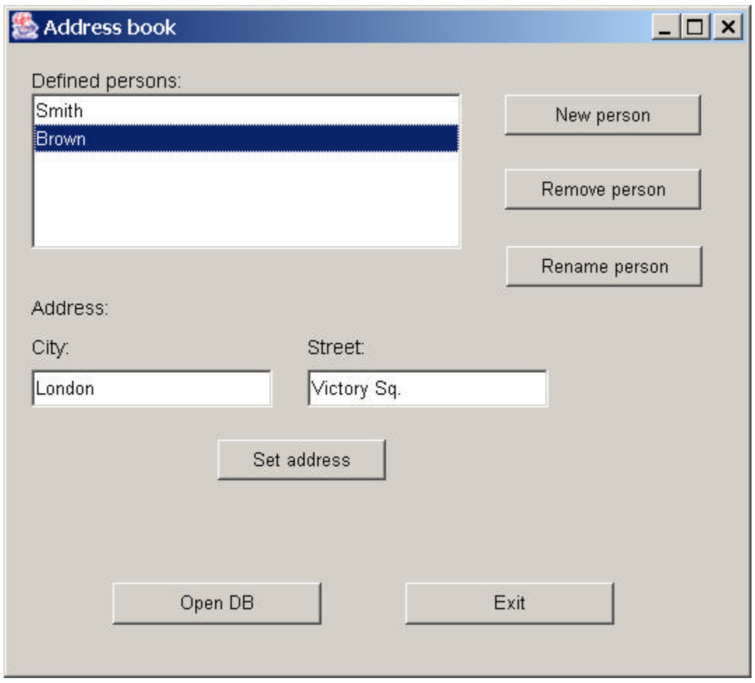


Fig. 38. The main window of the “Address book” test application

This application provides a GUI-based interface to manipulate the Person objects and their Address properties (see Fig. 38). A person’s *name* is being set at object’s creation time, when also an (initially empty) *Address* object, connected by a regular reference is created for it. The *Person* objects can be added and removed. Their addresses and names can be modified through a separate option. In effect, the following dependency kinds occur in this application:

- **Database lookup dependency**: after opening the provided federated database, the lookup of a database “People” occurs.
- **Database scan dependency**: the “People” database is scanned for the instances of the class *Person*.

- **Persistent objects' operation call**: needed to retrieve the strings denoting person's *name*, as well as the *city* and *street* of its address, as well as to update those fields (encapsulated by appropriate operations).
- **Persistent objects' direct access**: for the sake of example, the functionality allowing to modify a person's *name* uses direct access to persistent object's attribute rather than a dedicated operation.

"Department – Employee" application

This application is provided to test proper extraction of some more specific schema constructs used here, as well as another kind of dependency, not occurring in the previous example. Similarly like the previous application, "Department-Employee" uses a GUI interface to create / retrieve / update / delete its entities, which in this case are objects of *Department* and *Employee* (inheriting from *Person*) classes. The following functionality is provided (cf. Fig. 39):

- Adding and removing departments and the ability to select them and browse their employees.
- Adding employees (with default salary) to the selected department and removing them.
- Updating the employee's salary.
- Reassigning an employee to another defined department.

In case of the removal of non-empty department, a new pseudo-department, called "UNNASSIGNED" is created and bound within the "Persons" database as a named root object (of the name "UNASSIGNED"), in order to make the employees of removed department still available.

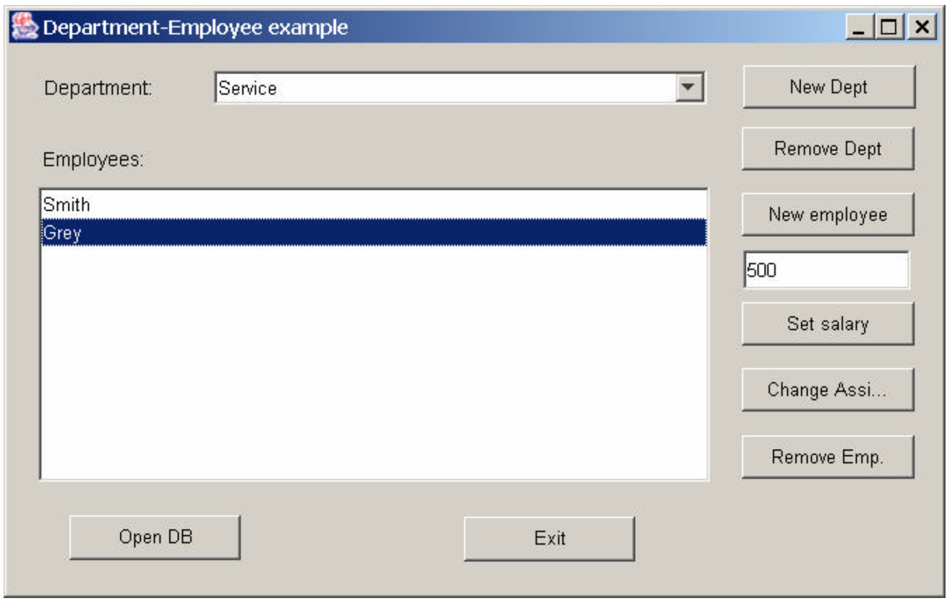


Fig. 39. The main window of the “Department-Employee” test application

Moreover, as can be seen in Fig. 37, Employees are connected with Departments using the ODBMS relationship mechanism rather than a plain reference. Another schema construct not occurring in the first application is the root object, registered within the scope of the “Persons” database.

Summing up, the following dependency kinds are expected to be discovered during testing of this application:

- **Database lookup dependency** – as in case of the first application.
- **Database scan dependency** – scan for the “Department” class instances.
- **Persistent object’s operation call** – the most common kind of dependency, similarly like in the first application.
- **Root object lookup**: lookup of the *Department* class root object called “UNASSIGNED”, performed within the scope of the “Persons” database.

D. List of figures