Implementation of a Framework supporting Web-based Information Modelling

Diploma thesis by
Claudia Ignat

Swiss Federal Institute of Technology Zurich
Institute for Information Systems
Global Information Systems Group

Professor Moira C. Norrie
Supervised by Adrian Kobler

June 2000

Acknowledgment

I want to thank my supervisor, Adrian Kobler, for all explanations and discussions we had, ideas for ingenious solutions. I want to thank him for all support he has given me, never hesitating to help me.

Also, I want to thank Professor Moira C. Norrie for offering me the possibility to work for an interesting subject and in a well trained group.

My thanks go also to all Global Information Systems members for being so nice with me and for creating a collegial atmosphere.

Finally, but not last, I want to thank Professor Kalman Pusztai from Technical University of Cluj-Napoca for offering me the chance to do my diploma project at ETH Zurich.

Contents

Chapter 1

Introduction

1.1 Problem formulation

It is increasingly the case that web browsers are not only used for browsing the world-wide web, but also serve as user interface for all kinds of web-applications. For example, web-applications providing access to databases typically use *forms* for user interaction making it possible for people, for instance, to update information. In this report, we look at web-applications enabling users to manage information in a flexible way, i.e. to categorise information according to user-defined criteria, and to organise, respectively, to structure information. We argue that such applications must support the activity of identifying entities, classifying entities, creating relationships between entities, and adding properties to entities without the need of changing the schema of the underlying database.

The goal of the project was to implement a framework which can be used to develop Web based applications supporting Rapid Information Modelling (RIM). By *information modelling* we understand the activity of classifying and structuring information about a specific domain on a conceptual level. The term *rapid* means that the construction of information needs is done at the conceptual level without the necessity of defining a data model for information organisation.

For implementing our framework, we have used the eXtreme Design XD framework and some ideas of conceptual modelling for supporting RIM. We have used the generic object model OM as an expressive data model for supporting the conceptual design modelling process. As in OM model, we defined a clear distinction between role modelling, denoting the activity of classifying objects, and typing, denoting the activity of defining object properties. By supporting rapid information modelling, we give the user the opportunity to represent concepts on a specific domain, to create categories for classifying these concepts, build relationships between these concepts and add properties to these concepts.

Further, using this framework, it is possible to analyse information and generate schema definitions to be used by a database application. In our case, we have generated schema definitions which can be used by OMS Java, i.e. the framework is capable of creating data definition language (DDL) statements as well as data manipulation language (DML) statements.

1.2 Overview

Chapter 2 gives an overview of conceptual modelling and briefly presents the database application design process. Also, in this chapter there is described the generic Object Model OM.

Chapter 3 gives a description of eXtreme Design (XD) meta model and presents the XD framework.

Chapter 4 introduces the notions of rapid information modelling, presents the architecture of the project and describes, in details, classification and typing levels.

Chapter 5 presents some implementation issues concerning our framework and the application developed based on this framework. Also, there is given an overview of the functions offered by the framework and by the application. Further, we describe the necessary steps for application installation. Finally, we present some open issues.

Chapter 6 describes some issues for extending the framework and presents the conclusions for this project.

Chapter 2

Conceptual Modelling and Design Process

2.1 Data modelling concepts

Only raw data do not constitute information. In order to be useful in the process of inferring information, data must be organised in a certain way. The relationship between data, information and knowledge can be stated as follows: "Information is data about a specific subject or event, organized to convey or communicate knowledge. That is, information is data that is organized to tell us something" [Mod92]. By data modelling, the organisation of data must fullfill two conditions:

- to represent as appropriate as possible the real world
- to be adaptable for representation and manipulation on a computer.

These two conditions are sometimes opposite.

Data modelling supposes the identification of the most important characteristics of data which reflect the essence of their meaning.

"A **data model** is a collection of concepts that can be used to describe a set of data and operations to manipulate data"[BCN92]. If a data model describes a set of concepts from a given reality, then this data model is called **conceptual data model**. The concepts in the data model are described through linguistic and graphic representations [BCN92].

Elmasri and Navathe [EN94] classify data models according to the following categories:

- **high level** or **conceptual data models** which use concepts such as entities, attributes and relationships, such that a user can perceive data in an easy way. An *entity* represents a real world object or concept, such as `course`, that is stored in the database. An *attribute* represents some property of interest that further describes an entity,

such as course's `name`. A *relationship* among two or more entities represents an interaction among entities, for example `taken_by` relationship between a student and a course, representing a course taken by a student. **Object-oriented models** are frequently used as high-level conceptual models, especially in the software engineering domain.

- **low level** or **physical data models** which describe how data is stored in the computer by representing record formats and orderings and access paths which make the search for particular database records efficient.

- **representational** or **implementation data model** which is situated between the previous two extreme data models and which hides some details of data storage, but can be implemented on a computer system in a direct way. Such data models are relational, network and hierarchical.

According to strong typing criteria, data models can be divided into [Dol98]:

- **strong typed** data models in which data has to belong to a certain category. Data which naturally does not conform to any of the existing categories will be forced to belong to a certain category.

- **weakly typed** data models in which there is not made any supposition concerning the categories. Categories are allowed only if they are useful. Individual data can exist by themselves or can be related to other data. Information concerning categories, if there exist, is treated in the same way like information about data.

Strong typed data models have the disadvantage of lacking flexibility, which makes the representation of subtil semantical distinction difficult. Let us consider the category `Employee`, representing all employees in a company. In a strong typed data model, this category must be homogeneous, that is, all objects belonging to the category must have the same properties. But, married employees have different properties compared to unmarried employees. Suppose that for an element of the category there are properties such as `name`, `address`, `salary`. For example, the married employees must have an additional property `spouse_name`. In a strong typed data model, a married employee, in order to be classified in the `Employee` category, will loose the additional property `spouse_name`. So, we cannot distinguish if the employees in the category `Employees` are married or they are single. In a weakly typed data model, because categories are heterogeneous, a married employee can be classified in the `Employee` category by maintaining the additional property.

Strong typed data models have the advantage that data properties can be abstracted and examined in terms of the categories they belong to. The elements belonging to the categories have the same properties, so that the properties names can be associated to the category.

By the restriction that all objects belonging to a category must have the same properties, strong typed data models are less expressive in modelling real world situations.

The models mainly used in DBMS are strong typed, we will see that eXtreme Design uses weakly typed data models.

"A data model is a collection of mathematically well defined concepts that help one to consider and express the static and dynamic properties of data intensive applications"[BMS84]. *Static properties* of applications refer to objects, their properties and relationships among objects. *Dynamic properties* refer to operations on objects and their properties and relationships among operations. Also an application can be characterized by *integrity constraints* over objects and operations. Constraints are rules used to define static and dynamic application properties that are not conveniently expressed using the object and operation features of a data model. A constraint usually relates static and dynamic properties. For example, prevents an update that would produce an invalid database state.

A data model is composed of two parts [Dol98]:

- a set of structuring rules which express the static properties of data model and are defined in a DBMS by DDL (Data Definition Language). The structures can be expressed by specifying
  - permitted objects and permitted relations between objects using generic rules of definition
  - not allowed objects and relations between them, operations called *constraints*.
- a set of allowed operations on data which express dynamic properties of the data model and which are expressed in a DBMS by DML (Data Manipulation Language).

What is common to all data models is a collection of primitive abstraction mechanisms: classification, aggregation and generalization.

Abstraction consists in neglecting some irrelevant aspects and concentrating on properties which present interest according to a point of view. An elementary form of abstraction is *typing*. We distinguish some distinct views for the definition of type [Nor95]: "Some philosophers consider general concepts as being defined in terms of a type which gives necessary and sufficient conditions for determining whether or not a particular general concept applies to a particular entity. Others consider that a general concept is

associated with a group of entities but that it may not be possible to state necessary and sufficient conditions to determine membership of this group". In database systems, both views prevail. The classification can be performed by the user which will insert entities in some collections representing classifications. Then, for describing the representations of the entities, there are defined forms that can be taken by these entities in terms of a set of structural and behavioural properties.

But, in object data models, there is a distinction between typing and classification [Nor95]. The two notions are linked in the sense that similar entities will have similar representations, so "an entity category will have an associated type that describes the form of representation of its members".

In the definitions that follow we try not to break this distinction between typing and classification.

Abstraction can be done on more levels. The types on one level constitute objects for the superior level of abstraction. In this way there can be obtained *hierarchies of types*. When the notions for generalization and aggregation will be presented, we will give such examples of hierarchies.

Classification abstraction is the process of grouping objects with common properties into categories. We can say that "classification is a form of data abstraction in which an object type is defined as a set of instances" [BMS84]. This is not contradictory to the separation of typing and classification in case of object models, because a category has associated a type for its members. The grouping of a set of students in the generic type `student` associated to the category `Students` is done by classification. *Instantiation* is the opposite of the classification process. Classification establishes an `instance-of` relationship.

If classification is a relationship between a type and its instances, generalization and aggregation are relationships between types [BMS84].

Generalization abstraction is a form of abstraction that relates a type to more specific ones. The generalization relation between types, also called `is-A` organizes types into a generalization hierarchy. In Figure 2.1 there is shown that the type `person` is a generalization of types `student` and `teacher`. The opposite process for generalization is *specialization*.

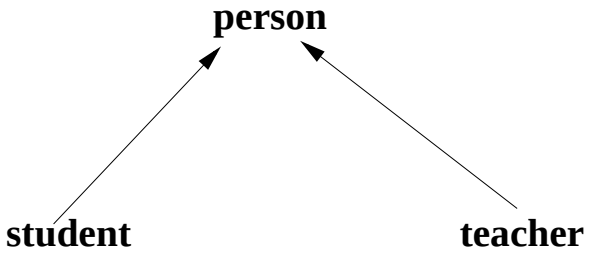

Figure 2.1: Generalization hierachy

Aggregation is a form of abstraction by which an object is constructed from its component parts. Aggregation is also called `part-of` relation. In Figure 2.2 there is shown that the type `teacher` is constructed by aggregation of types `name`,`address`, `title`, `office`. Like generalization, aggregation can be repeated on different levels resulting in a hierarchy of aggregation. For the given example, the type `address` is an aggregate type having as constituents `street`, `number`,`city`.

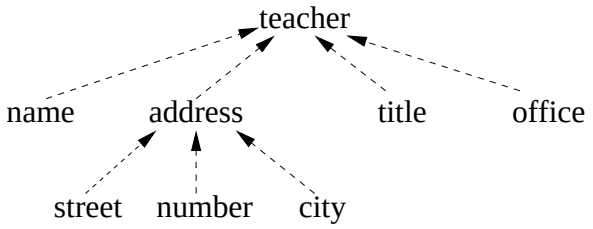

Figure 2.2: Aggregation hierachy

There exist three types of relationships between two sets of entities S_1 and S_2:

- **1:1** relation when to an entity in S_1 there corresponds an unique entity in S_2 and conversely (wife-husband relationship)
- **1:N** when to an entity in S_1 there correspond one or more entities in S_2, but to every entity in S_2 there corespond an entity in S_1 (father-son relationship)
- **M:N** when to an entity in S_1 there correspond one or more entities in S_2 and conversely (friend-friend relationship)

2.2 Database Application Design

The development cycle of a complex software applications includes stages of analysis, design and implementation.

The **analysis** phase addresses the question of "what the system is intended to do"[DT93]. This stage will produce a set of specifications and requirements.

The **design** phase addresses the question of "how the system will carry out its required function"[DT93]. This stage will produce a design document which acts as the *plan* for implementation.

If system analysis is the process of separating the system into parts for "studying, examination and repair or replacement", design is "that set of processes and activities whereby the specifications as to how that data is to

be gathered, maintained, processed into information and made available to the use are articulated" [Mod92].

Implementation involves coding and testing the individual modules and integrating and testing the system with other systems.

Activities related to database applications include: requirements analysis, application design, implementation, testing and validation, monitoring and maintenance.

2.2.1 Database Design

Elmasri and Navathe [EN94] propose that the database design process consists of the following main steps:

- **Conceptual database design**

  Here there are two parallel activities:

  - *conceptual schema* design which produces a schema independent of a specific DBMS, so there must be used a high-level data model (such as the ER or the OM model which is described in section 2.2.3) for specifying the schema.
  - *transaction design* for designing the characteristics of known database transactions in a DBMS-independent way.

- **Data Model Mappping** also called **Logical Design**

  The conceptual schema from a high-level data model is transformed into a data model of a chosen DBMS.

- **Physical database design**

  In the physical database design stage, the physical storage structures are specified.

2.2.2 Conceptual design

Conceptual design begins by specifying requirements and ends by generating a conceptual schema. The *conceptual schema* describes at a higher-level of abstraction the structure of the database, independent of the particular DBMS system that will be used to implement the database; so, it will describe the information content of the database and not the storage structure for managing information.

Successful results of conceptual design can be obtained only through a cooperation of designers with database users who are responsible for describing requirements and explaining the meaning of data. Conceptual design and conceptual data models must provide an easy understanding for users who do not have much technical knowledge of database systems. This means

that one can use the conceptual model to specify concepts of the real world [SW94]:

- **categorization** is a form of abstraction by which humans are grouping things into categories.
- **grouping** by which a set of things is divided into smaller sets of things which are relevant to some subtopic within an overall topic.
- **shielding from details** by which some things are described in terms of other things, which at their turn can be detailed.
- **aggregation and ordering** by which something is composed of other things. Aggregation can also be sequenced.

"The most important advantage of conceptual design shows up during the operation of database, when conceptual model and its documentation ease the understanding of data schemas and of applications that use them and this facilitate their transformation and maintenance" [BCN92].

The typical **activities** of conceptual design are the following [BCN92]:

- **Requirements Analysis**

  In this step the designer studies the requirements, eliminates the ambiguities (imprecise or incorrect descriptions of reality), classifies requirements.

- **Initial conceptualization**

  In this step there are produced the first concepts for representing the conceptual schema, i.e. there are created abstractions to be represented as entities, relationships and generalizations.

- **Incremental conceptualization**

  This is the central activity of conceptual design, by which a draft schema is step by step refined into the final conceptual schema, using strategies such as: *top-down strategy*(concepts are progressively refined), *bottom-up strategy*(starting from elementary concepts more complex concepts are built out of them), *inside-out strategy* (there are fixed the most important concepts, then there are found the concepts that are conceptually close to the starting concept and then navigate toward the most distant ones) and *mixed strategy* (top down partitioning of requirements, bottom up integration using a skeleton schema).

- **Integration**

  Consists of merging several schemas and producing a new global representation of all of them, by determining the common elements of

different schemas and discovering conflicts (different representations of the same concepts) and interschema properties (constraints among different schemas).

- **Restructuring**

  Consists of measuring and improving the quality of the product obtained (completeness, correctness, minimality, expressiveness, readability, self-explanation, extensibility, normality).

As mentioned before, the parallel activity to the conceptual schema design in the conceptual design database phase is the **transaction design**. A transaction must be specified in a conceptual and system-independent way, by identifying the input and output data and the internal functional flow of control of the transaction. "*Transactional modelling* involves the design and specification of structural and behavioural properties of transactions" [BCN92].

2.2.3 Generic Object Model OM

To support the conceptual design modelling process, we need an expressive data model.The generic object model OM is suitable for conceptual design because it can be used for all stages of the database development cycle and offers a rich set of concepts for classification and typing.

In terms of typing, OM model supports the main object-oriented concepts, a brief description of which is given in the following paragraphs.

An object in an object-oriented environment has a unique **identifier** that is independent of the values of its attributes. When created, an object will be assigned an unique identifier and will be referred to using this identifier during its lifetime. The internal structure of an object consists of two basic components: attributes and methods.

Attributes describe the state of the object. An attribute consists of two parts, an *attribute name* and an *attribute value*. The value part of the attribute is itself an instance object and can be either a simple object or a complex object. *Simple objects* consist of primitive types (integers, characters, ...) and *complex objects* can consist of tuples, sets, lists, arrays. A complex object can also be a recursive structure of some or all of these elements. [Atk89]

Methods describe the behaviours associated with an object. They represent the actions that can be performed by an object.

Encapsulation or **information hiding** is the property of an object that its attributes can only be accessed by the methods of the object.

Objects communicate with each other by means of **message passing**. The message interface defines a clear interface between the object and the rest of its environment. The protocol of message passing involves two parties,

a *sender* and a *receiver*. A message has associated a method within an object.

Class is defined as "a description of one or more objects, describable with an uniform set of attributes and services". [CY91] "A *service* is a specific behaviour that an object is responsible for exhibiting". [CY91]

Inheritance is defined as a "mechanism for expressing similarity among classes, simplifying definition of classes similar to ones previously defined"[CY91]. Inheritance introduces the notions of *generalization* and *specialization*. Classes with common properties are grouped into a superclass representing the generalization of the subclasses. A subclass of a given class represents a specialization of it.

Polymorphism is an important feature of Object-Oriented Systems which "allows one to develop systems where different objects could respond in a different way to the same message" [DT93].

For example, in a university we want to have an information system capable of managing `student` and `teacher` data. So, we can define a type hierachy as shown in Figure 2.3. We assign to a `person` properties such as `name`. Additionally, we assign to a `student` the property `year` of study and to a `teacher` properties `title`, `salary` and `office`.

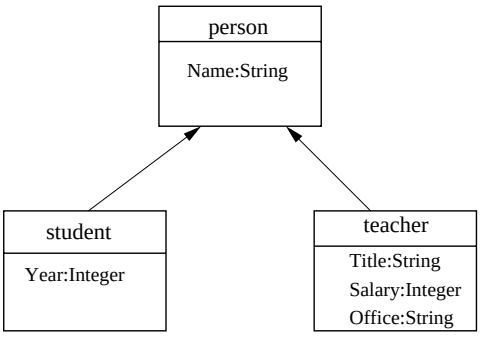

Figure 2.3: Typing hierarchy for University example

Further, we want to classify persons into students and teachers.

The OM model distinguishes between the *classification* of entities into categories according to general concepts of the application domain and the description of representation of entities within database in terms of *types* [Nor95].

In reality, a person can be classified as a student and as a teacher simultaneously. In this way, a person will play the role of a student and of a teacher at the same time. The OM model allows objects to have different types simultaneously. So, the same person object can be dressed with the types `student` and `teacher`. Such role modelling cannot be supported by

most object-oriented data models.

In our example, as shown in Figure 2.4, role modelling is achieved by inserting objects into collections `Persons`, `Students` and `Teachers` and it is possible that an object belong to more than one collection at the same time. For instance, if an object is member of the collection `Students` and `Teachers` simultaneously, it gains the attributes `Name`, `Year` if it's viewed through collection `Students` and the attributes `Name`, `Title`, `Salary`, `Office` if it's viewed through collection `Teachers`.

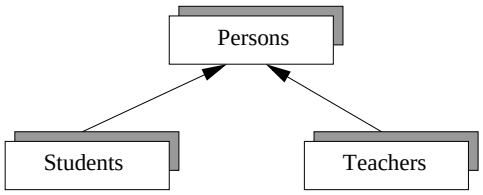

Figure 2.4: Classification

Since OM is a data model, it consists of *data structures*, *operations* for data retrieval and update and *integrity constraints* which will be discussed in more detail in the following paragraphs [SKN98].

Data Structures

The OM model provides support for collections of objects and associations (binary collections).

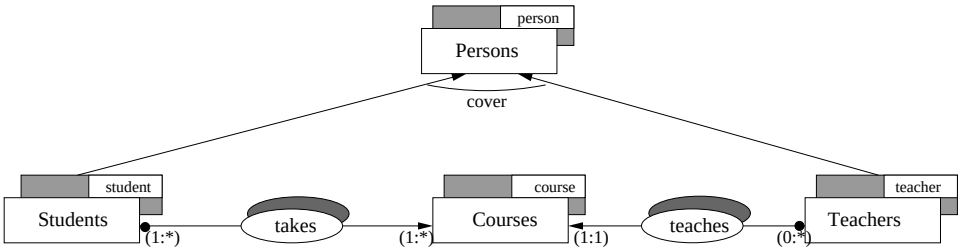

Figure 2.5: A schema in the OM model

In Figure 2.5 there are represented the collections `Persons`, `Students`, `Teachers` and `Courses` and the associations `takes` between collection `Students` and `Courses` and `teaches` between `Teachers` and `Courses`. In the shaded region of each box denoting a collection, the member type of the collection is given. That means that an object gains the member type of the collection whenever it is accessed through this collection.

For each association we can specify cardinality constraints. In Figure 2.5, there are cardinality constraints between the collections `Students`

and `Courses` and between `Students` and `Teachers`. A `student` must take at least 1 course *(1:*)*, a `course` must be taken by at least 1 `student` *(1:*)*, otherwise it will be not taught, a `teacher` can teach 0 or more courses *(0:*)* and a `course` must be taught by exactly one teacher *(1:1)*.

A collection, like any object, has a unique object identifier. An association contains pairs of object identifiers refering to the source and target objects.

Algebra

OM specifies a set of operations over collections and binary collections.
Operations on collections

- *Union*

  $\cup : (coll[t_1], coll[t_2]) \rightarrow coll[t_1 \sqcup t_2]$

  A union operation forms a collection containing the elements of the two specified collections. The result collection will have a member type which is *least common supertype* of t_1 and t_2 denoted by $t_1 \sqcup t_2$. The least common supertype of two types is defined as the intersection of the set of properties of the two types. [NW99]

- *Intersection*

  $\cap : (coll[t_1], coll[t_2]) \rightarrow coll[t_1 \sqcap t_2]$

  An intersection operation forms a collection containing those elements common to the two specified collections. The result collection will have a member type which is *greatest common subtype* of t_1 and t_2 denoted by $t_1 \sqcap t_2$. The greatest common subtype of two types is defined to be the union of the set of properties of the two types. [NW99]

- *Difference*

  $- : (coll[t_1], coll[t_2]) \rightarrow coll[t_1]$

  A difference operation forms a collection containing those elements of the first operand collection that are not members of the second operand collection. The member type of the resulting collection is the member type of the first operand collection.

- *Cross Product*

  $\times : (coll[t_1], coll[t_2]) \rightarrow coll[(t_1, t_2)]$

  The cross product of two collections returns a binary collection containing all combinations of an object of the first operand collection with an object of the second operand collection.

- *Flatten*

$\pm : (coll[coll[t]]) \rightarrow coll[t]$

The flatten operator takes a collection of collections of the same member type and flattens them to a collection of values of that member type.

- *Selection*

  $\% : (coll[t], t \rightarrow bool) \rightarrow coll[t]$

  A selection operation on a collection forms a collection containing those elements of the collection that satisfy a given predicate. The predicate is represented by a function that maps each element of the collection to a boolean value.

- *Map*

  $\alpha : (coll[t_1], t_1 \rightarrow t_2) \rightarrow coll[t_2]$

  A map operation on a collection applies a function to each element of collection and forms a collection of the results.

- *Reduce*

  $\oplus : (coll[t_1], (t_1, t) \rightarrow t, t) \rightarrow t$

  The reduce operator is used to give aggregate operations such as sum, maximum, average over collection of values. It has three arguments: the collection $coll[t_1]$ over which the aggregate function is executed, the aggregate function with a signature $(t_1, t) \rightarrow t$ and an initial value of type t.

- *Cardinality*

  $\# : coll[t] \rightarrow \text{int}$

  The cardinality of a collection is the number of elements of that collection.

Operations on binary collections

- *Domain*

  $domain : coll[(t_1, t_2)] \rightarrow coll[t_1]$

  The domain operation takes a binary collection and forms a collection of all the objects that appear as the first element of a pair of objects $< oid_1, oid_2 >$ belonging to the binary collection.

- *Range*

  $range : coll[(t_1, t_2)] \rightarrow coll[t_2]$

  The range operation takes a binary collection and forms a collection of all the objects that appear as the second element of a pair of objects $< oid_1, oid_2 >$ belonging to the binary collection.

- *Inverse*

  $inv : coll[(t_1, t_2)] \rightarrow coll[(t_2, t_1)]$

  The inverse operation swaps the first and the second elements of the pairs contained in the collection and returns them in a new binary collection.

- *Compose*

  $\circ : (coll[(t_1, t_2)], coll[(t_2, t_3)]) \rightarrow coll[(t_1, t_3)]$

  The composition operation combines those binary objects of the two collections where the second object in the first binary object $< oid_1, oid_2 >$ appears as first object of the second binary object $< oid_2, oid_3 >$, and the result contains binary objects of the form $< oid_1, oid_3 >$.

- *Closure*

  $closure : coll[(t_1, t_2)] \rightarrow coll[(t_1, t_2)]$

  The closure of a binary collection is the reflexive transitive closure of a relationship represented by the binary collection.

Constraints

The OM model defines the subcollection, cover, disjoint, partition, intersection and cardinality constraints. The *subcollection constraint* means that each object in a collection is also member of its supercollection. The *cover constraint* means that each object of the supercollection appears in at least one of the subcollections which are under this constraint. The *disjoint constraint* means that the set of the subcollections under this constraint do not share any object of the supercollection. The *partition constraint* is equivalent to the combination of *cover* and *disjoint* constraints. The *intersection constraint* means that all common objects of the supercollections of a collection are members of that collection. The *cardinality constraint* restricts how often an object may be contained in an association.

The OM model can be used for the stages of database development. In the *conceptual database design* stage, as we have seen, we express knowledge about objects, semantic grouping of objects, associations between objects, as shown in Figure 2.5.

During *data model mapping* phase, the conceptual schema is transformed to the data model of the chosen DBMS. For example, in the case of OM, this is best done in terms of DDL statements.

```
type person
(
  name:  string;
```

```
);

type student subtype of person
(
  year:  integer;
);

type teacher subtype of person
(
  title:  string;
  office: string;
  salary: integer;
);

type course
(
  name:        string;
  identifier:  string;
  room:        string;
);

collection Persons:    set of person;
collection Students:   set of student;
collection Teachers:   set of teacher;
collection Courses:    set of course;
collection take:       set of (student, course);
collection teach:      set of (teacher, course);

constraint Students subcollection of Persons;
constraint Teachers subcollection of Persons;

constraint (Students and Teachers) cover Persons;

constraint take association from Students (1:*) to Courses (1:*);
constraint teach association from Teachers (0:*) to Courses (1:1);
```

The population of the database is described in terms of DML statements. For example, for creating an object of type `teacher`, we can specify the following DML statements:

```
create object teacher1;
dress teacher1 as teacher values
(
```

```
  name = "Moira Norrie";
  title = "prof";
  office = "IFW D45.1";
  salary = "10000";
);

insert teacher1 into collection Teachers;
insert teacher1 into collection Persons;
```

Chapter 3

eXtreme Design

One aspect of eXtreme Design (XD) is that it "facilitates rapid information modelling (RIM) by making it possible to specify and implement concepts of the application domain on a very high level of abstraction" [Kob00].

eXtreme Design combines concepts from conceptual modelling and object-oriented software construction. The eXtreme Design approach comprises the XD meta model and the XD algebra and it is supported by XD framework. So, in the following sections, we give a description of the XD Meta Model and present the XD Framework.

3.1 eXtreme Design Meta Model

Figure 3.1 shows the XD meta model which is divided into 3 main parts corresponding to the activity of *classification*, *structuring* and *typing*.

The meta model is defined in terms of the OM model, making it possible to easily customize the meta model by adding collections and associations.

Further, the set of object attributes and methods is not fixed, they can be added and removed at runtime. Because of this characteristic, an object in the meta model "can be regarded as representing an object in the context of the object-oriented paradigm on a higher level of abstraction. We call it therefore a *meta object*"[Kob00].

There exists an XD Algebra consisting of algebraic operations for all three parts of the XD meta model. There are defined algebra operations for object groups, for object relationships and facts, and for object properties. [Kob00]

3.1.1 Classification

The activity of classification corresponds to role modelling. Here, we specify the roles the objects play in application domain. `Objects` can be classified to be members of `Object Groups` as shown in the Figure 3.1 by the as-

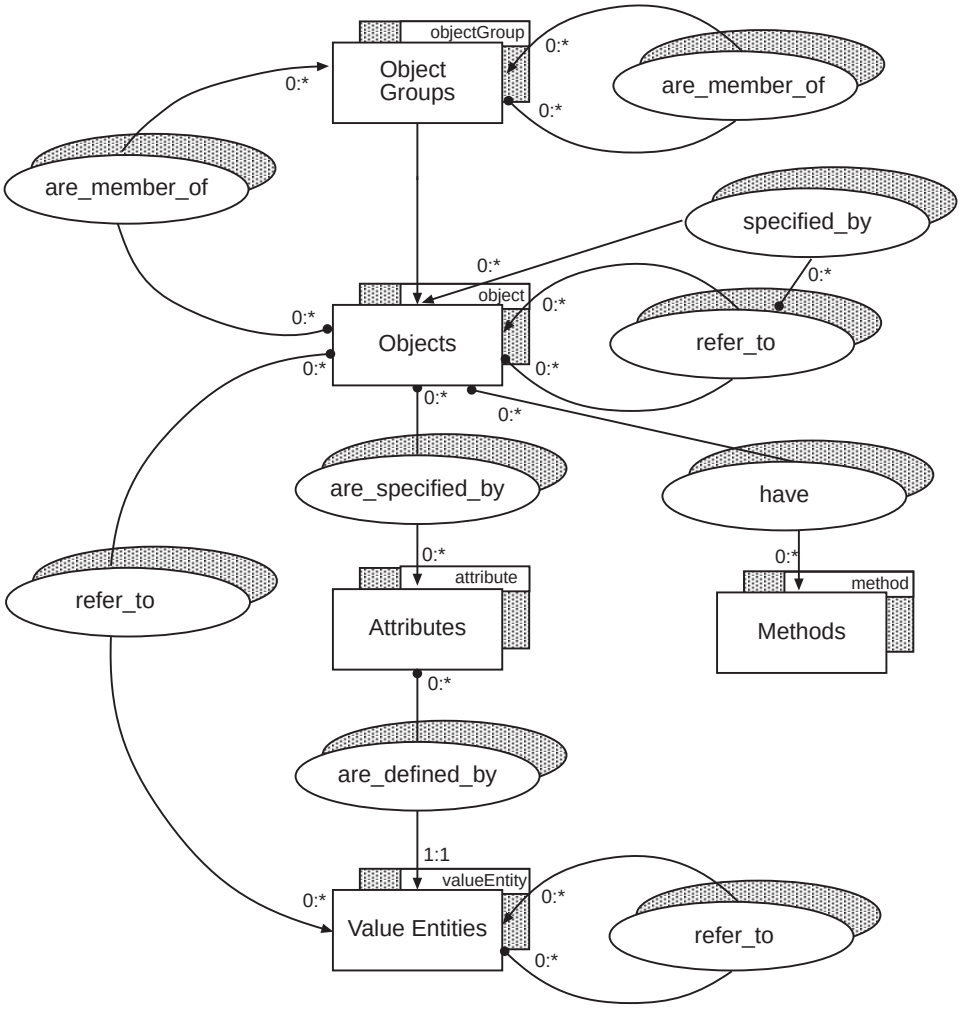

Figure 3.1: A schema in the OM model

sociation `are_member_of` between `Objects` and `Object Groups`. An object group can be member of other `Object Groups`, as shown by the association `are_member_of` between `Object Groups`. Objects and object groups are represented in the same way, by a unique object identifier, but the semantic is different in the sense that object groups are the equivalent of categories as opposed to objects which represent simple entities.

Objects can be created by using the following XD DML statements:

```
create object teacher1;
create object student1;
create object student2;
create object course1;
```

With the following XD DML statements we can create object groups:

```
create object Persons;
create object Students;
create object Teachers;
create object Courses;
```

For inserting objects into collections we use the following DML statements:

```
insert into Persons:  [teacher1, student1, student2]
insert into Teachers: [teacher1];
insert into Students: [student1, student2];
insert into Courses:  [course1];
```

3.1.2 Structuring

Structuring refers to defining relationships between objects. In conceptual modelling, relationships can either exist between instances of concepts called *instance relationships*, or they can exist between generic concepts called *generic relationships* [DT93]. For example, an instance relationship is *Moira Norrie teaches Object-Oriented Databases* and a generic relationship is *teacher teaches course*.

Further, the relationships can be classified into *hierarchical relationships* and *nonhierarchical relationships*.

Hierarchical relationships can be classified into [DT93]:

- *is_a relationships* which are relationships between a general concept and a more specialized concept. A `Student` is a specialization of `Person`.
- *instance_of relationships* describe the relationships between an instance concept and a generic concept. *Moira Norrie is a Teacher* is an example of such a relationship.

- *composition or part_of relationships* describe the relationships between a composite concept and its component concepts. For example, it can be stated that a `HumanBody` consists of parts `Head`, `Legs`, `Arms` and so on.

- *membership_of relationships* define concepts that populate a set, linking individual instances of concepts to another generic concepts. For example, students which are of a certain nationality. Suppose the discount rate for students being of a certain nationality, Swiss for example, has to be changed. Then the discount rate can be altered for the generic concept Swiss and not for every instance of students of that nationality.

Non-hierarchical relationships link two or more concepts in an arbitrary non-hierarchical manner. Each such relationship is characterized by the number of instances of one concept that have a relationship for each instance of the other concept. There exist *one-to-many* and *many-to-many* relationships.

In XD model, the objects can refer to other objects by using the association `refer_to`.

How can we represent the previous enumerated relationships using XD model?

For example, in case of a relation between an instance of `Teachers` and an instance of `Courses`, the following DML statements can be specified:

```
create object teacher1;
create object course1;
association(teacher1, course1);
```

Because object groups are represented in the same way as objects, is_A relationship is represented in the same way as a relationship between two objects:

```
create object Persons;
create object Students;
association(Persons, Students);
```

However, if an object is inserted into `Students`, it must be inserted also into `Persons`, so this must be explicitly done. An application which evaluates DML statements and which knows the semantic of relationships between object groups, can take into account the necessary operations which must be performed according to the specified relationships. For example, if this relation has the semantic of a subcollection relation, then, whenever an object is inserted into a collection, it must be inserted into its supercollections.

The instance_of relationships are specified by inserting that object into the corresponding object group:

```
create object Students;
create object student1;
insert into Students:[student1];
```

An example of part_of relationship is that a human body is composed of the parts: head, body, legs, arms. This can be described as follows:

```
create object HumanBodies;

create object myHead;
create object myBody;
     ...

create object humanBody;
assert(humanBody, head, myHead);
assert(humanBody, body, myBody);
```

Membership_of relationships can be represented in terms of methods.

One-to-many and many-to-many relationships can be easily represented. There is no restriction on how many times an object can appear on the first or on the second position in the specification of a relation. For example, the relationship between `Students` and `Courses` is a many-to-many relationship.

```
create object Students;
create object Courses;
create object student1;
create object student2;
create object course1;
create object course2;
insert into collection Students:[student1, student2];
insert into collection Courses:[course1, course2];
association(student1, course1);
association(student1, course2);
association(student2, course1);
```

The `specified_by` association between `refer_to` association and `Objects` collection can be used to characterize the `refer_to` association by assigning *facts* [Kob00]. This way, we can assign attributes and values to `refer_to` associations.

For instance, if we want to keep track of how many hours `course1` has been taught by `teacher1`, then we can create the fact

```
create object fact1;
assert (fact1, hours, 3);
```

and assign it to the corresponding `refer_to` association.

```
fact((teacher1, course1), fact1).
```

3.1.3 Typing

Typing refers to methods, attributes and attribute values. Attributes are linked to the object by `are_specified_by` associations and methods are linked to the object by `have` associations. Value entities are associated to attributes by `are_defined_by` association and to objects by `refer_to` association. Value sets can be specified by `refer_to` association between value entities.

So, to a meta object there can be assigned attribute objects as well as value objects. "An attribute object serves as a placeholder for grouping value entities in a similar way as group objects are used for classifying meta objects" [Kob00].

For creating an attribute object `name` for the meta object `person1` and assigning the value object `student1`, we can use the following DML statement:

```
assert(person1, name, student1);
```

For creating multi-value attributes, more than one value objects can be asserted to the same attribute object. For example, to assert two value objects `office1` and `office2` to the attribute object `office` of the object `person1`, one can use the following statements:

```
assert(person1, office, office1);
assert(person1, office, office2);
```

Method objects which are associated to meta objects can be specified either using a programming language such as Java or using one of the XD languages such as DML or the XD meta language (XDML) which is based on Oberon-0 [Wir96].

A *type* in the XD meta model is also a meta object and denotes a view to object properties. In the following example we create the type object `courseType` and the attribute `name` having the value `java.lang.String`. The attribute `name` has the *scalar type* `java.lang.String`. A scalar type, also called base type, primitive type or unstructured type, is a type whose elements are indivisible. As opposed to scalar types, there exist *structured types*. For example, `courseType` is a structured type. Further, we create the object `course1`. By *dress* operator, the object `course1` is " dressed " by the type object `courseType`. So, further operations on the object `course1` will take into account the fact that object `course1` is of type `courseType`. The attributes of object `course1` will be the attributes of object type `courseType`. The following DML statements create the object `course1`, dress it with the type `courseType` and assign it the scalar value `"Course1"` to the `name` attribute object.

```
create object courseType;
```

```
assert(courseType, name, "java.lang.String");
create object course1;
dress course1 as courseType values
(
  name="Course1";
);
```

3.2 eXtreme Design Framework

There exists a framework supporting the eXtreme Design meta model called **eXtreme Design Framework**. There are a lot of functions for managing the meta model provided as API for application development. For example, the following list gives an overview of some of these functions:

- creating, deleting objects, retrieving an object by its object identifier or by its alias
- creating and removing attributes of a given object, iterate over the attributes of an object
- creating values, iterate over the values of an attribute of a given object
- creating and deleting facts
- creating and executing methods
- inserting objects into groups, removing members from a group, iterating over the members of a group

The XD framework has been used for this project as a basis for developing the framework for rapid information modelling.

Chapter 4

Rapid Information Modelling

In this chapter, we will give a brief description of Rapid Information Modelling (RIM) and motivate the necessity for this kind of modelling by means of an example. Further, we will present the architecture of our framework and describe how classification and typing levels were approached in our framework.

By *information modelling* we understand "the activity of classifying and structuring information about a specific domain on a conceptual level, i.e. the organisation of information is such that people are able to gain *value* from their point of interest"[Kob00]. The problem is to satisfy various information needs for the same information domain called *information space*[Kob00].

The term *rapid* means that the construction of information spaces is done at the conceptual level without the necessity of defining a data model for information organisation.

A RIM application should manage information spaces for users, facilitating the organisation and navigation of information. Using a RIM application, the user is able to conveniently represent concepts of a specific domain as *information objects*, add properties to these information objects, build relationships between them, create categories for classifying such objects. Our framework can be used for developing such an application.

In what follows we will motivate the need for information modelling by means of an example. Let us consider again the university example. Consider the `Students` collection. Each student object which is member of this collection has associated properties such as `name` and `year` of study. Suppose that, after having the list of all students, we find out that 1% of the existing students not only study but also work. Thus, for these students, we decide to keep some extra information about their job: `address_of_work`, `phone_from_work`, `salary`.

In case of the OM model, each collection has associated a type, and an object belonging to that collection is dressed with that type. A solution

for the described problem would be to change the type `student` associated to the collection `Students` so that to reflect the extra information. In that case the type `student` would have associated the attributes `name`, `year`, `address_of_work`, `phone_from_work`, `salary`. But it does not make sense to have all these properties for the students who do not work. On the other hand, the type `student` associated to the collection cannot be changed at runtime. Another possibility would be to create a new collection `StudentsThatWork` with the associated type `studentThatWorks`. Type `studentThatWorks` has the attributes `address_of_work`, `phone_from_work` and `salary`. So, a student that works can be accessed both from collection `Students` and then is viewed as being of type `student` and from collection `StudentsThatWork` and then is viewed as being of type `studentThatWorks`.

So, there are problems when we want to add some attributes to an object belonging to a collection without needing to change the schema. The same problem arise when we want to delete some attributes of an object in a collection. For instance, suppose that some students teach courses. So, we would like to insert a student object into the collection `Teachers`. Collection `Teachers` has associated the type `teacher` having the attributes `name`, `title`, `office`, `salary`. So, for a student object inside the collection `Teachers` we would like to have only the attributes `name`, `title`, `salary`. Moreover, for a student there must be added the attribute `year` of study.

We argue the necessity to add or remove attributes of an object at runtime and also to model heterogeneous collections. Adding and removing attributes of objects at runtime can be realised by using the XD approach. Heterogeneous collections are collections in which the members have some common features, but not all their properties are the same. Heterogeneous collections can be implemented by associating not only one, but more types to a collection.

4.1 Architecture

The aim of the project was to model the *data model* used at client application level using the *meta model* from the server side.

For understanding what means "OM Instances" and "OM Objects" in Figure 4.1, it must be explained how OM objects are represented in OMS Java. An OM object has a unique object identity during its whole lifetime. In Java, a reference to an object is unique as long as the object is loaded in a Java Virtual Machine. In case of saving an object and then restoring, for example in a file, there will be created a new reference to it. So, it cannot be determined using the reference if the two objects (the one before saving and the one after restoring) are the same. As a solution for this problem, it was introduced the class *ObjectID*. Every instance of this class represents a unique persistent object identifier and is related to one *OM object*.

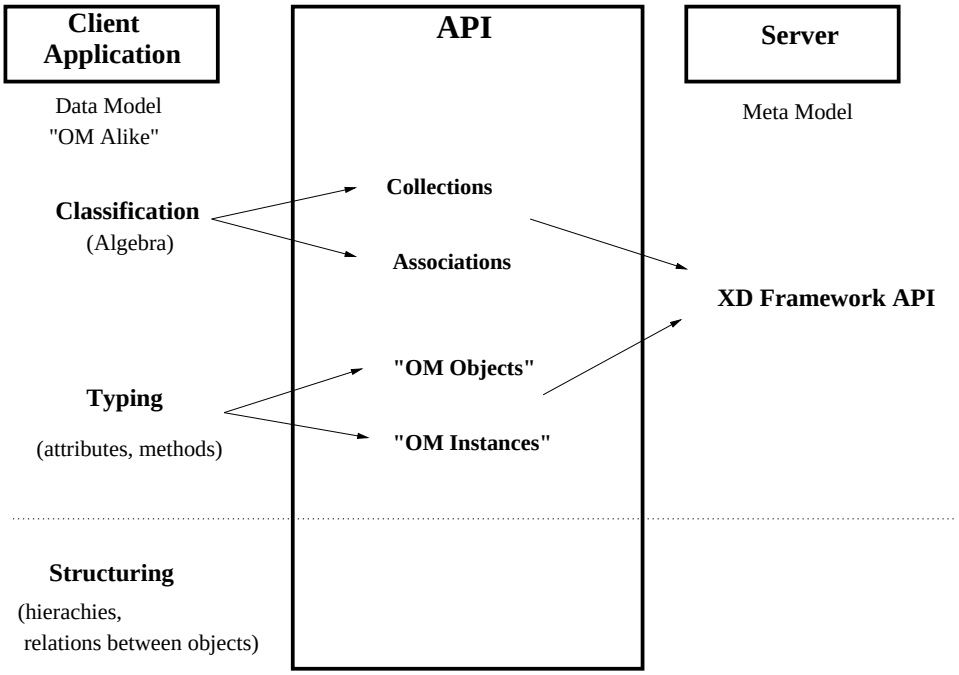

Figure 4.1: Architecture

As we have seen, an OM object supports role modelling, i.e. it can have more than one type associated to it. This is not supported by Java because the type definition of a Java class instance cannot be changed during runtime. As shown in Figure 4.2, the solution for this problem was that of splitting an *OM object* into two classes `OMObject` and `OMInstance`. Information about OM objects needed by OMS Java system are defined in instances of the Java class `OMType`.

According to XD meta model, there are three levels corresponding to the three activities: classification, typing and structuring.

The *classification* level used in the "OM alike" data model was realized by implementing collections and associations and their algebra. The collections developed in our framework are more complex than the ones used in the OM model, in the sense that they allow members of different types. That is, the collections are heterogeneous.

At the *typing* level, as the OM model, the framework allows an object to have different types simultaneously. As we have previously presented, the OMS Java solution for allowing simultaneously different types for the same object was the implementation of classes `OMObject` and `OMInstance`. By representing in Figure 4.1 "OM Objects" and "OM Instances" we wanted to highlight the fact that an object can be instance of different types.

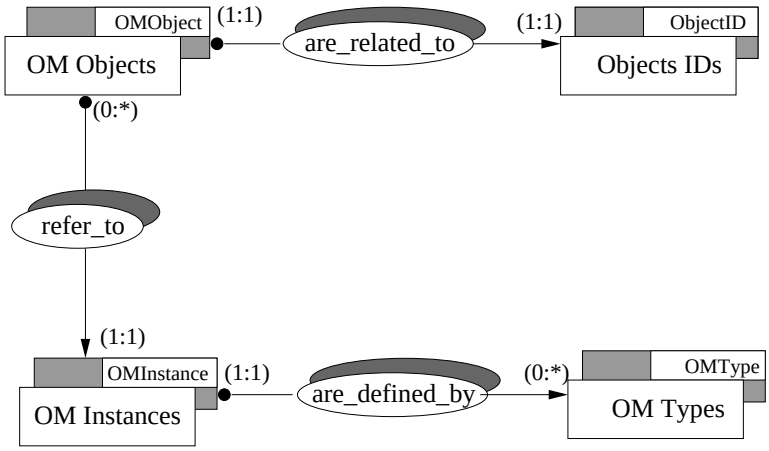

Figure 4.2: OM objects

Structuring refers to specifying object relationships. For example, in our framework we have developed subcollection relationship.

The framework was constructed based on XD Framework. It is an extension of XD Framework for dealing with heterogeneous collections and their algebra. This framework helps a developer to implement different functions for assisting the user in the process of classifying information.

In addition, it was also possible to analyse the data and generate a schema which can be used, in our case, for OMS Java (see section 5.2.3).

4.2 Classification

Classification was implemented by means of collections and associations or binary collections and their algebra.

A collection can contain objects of different types. Each object inside the collection has associated a type. Because a collection has associated different types, we can view the collection through any of these types. So, we can introduce the notion of *collection viewed through a type*.

We will denote a collection $\mathtt{C}$ having members of type $\mathtt{t_1}$, $\mathtt{t_2}$, ..., $\mathtt{t_n}$ by

$$\mathtt{C}\{\mathtt{t_1}, \mathtt{t_2}, \ldots, \mathtt{t_n}\}$$

Suppose A_i, $i = \overline{1,n}$, is the set of objects instances of types $\mathtt{t_i}$ belonging to collection $\mathtt{C}$. Then,

$$\mathtt{C}\{\mathtt{t_1}, \mathtt{t_2}, \ldots, \mathtt{t_n}\} = A_1 \cup A_2 \cup \ldots \cup A_n,$$

where A_i are disjoint sets, $i = \overline{1,n}$.

We want to define $\mathtt{C}(\mathtt{t_i})$ denoting the collection $\mathtt{C}$ *viewed through the type* $\mathtt{t_i}$. If an object member of $\mathtt{C}$ is not instance of type $\mathtt{t_i}$, it means that it does not belong to $\mathtt{C}(\mathtt{t_i})$. So, $\mathtt{C}(\mathtt{t_i}) = A_i$.

We will consider that if there is not specified a type to view the collection, then all members of the collection are accessed. Mathematically, we can specify this by one of the equalities:

$$\mathtt{C} = A_1 \cup A_2 \cup \ldots \cup A_n$$

$$\mathtt{C}\{\mathtt{t_1}, \mathtt{t_2}, \ldots, \mathtt{t_n}\} = A_1 \cup A_2 \cup \ldots \cup A_n$$

Also, we will adopt some graphical representations.

For representing a collection together with its members, we will use the representation in Figure 4.3. The rectangle in the figure contains two parts. The first part contains the name of the collection and in the second part, the object identifiers of the collection members are listed. The two parts are separated by a sharp line.

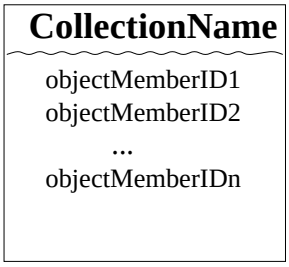

Figure 4.3: Collection Representation

Further, for representing an object instance of a type `typeName` together with its attributes we will use the representation in Figure 4.4. The rectangle in the figure comprises two parts: in the first part, the object identifier is given and in the second part, there are listed pairs of the form attribute=value. The two parts are separated by a straight line. The small rectangle in the back represents the name of the type the object is dressed with. Note that the object can have some missing or extra attributes compared to the attributes of the type. When representing it like this, there will be listed only the existing attributes of the object which belong also to the type.

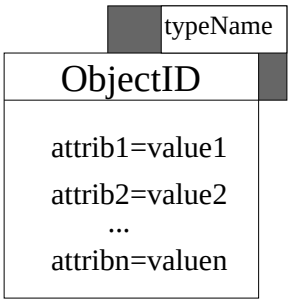

Figure 4.4: Object Representation

Let us give an example for illustrating the notion of collection viewed through a type.

Suppose collection `Teachers` contains three objects represented by their identifiers OID1, OID2 and OID3. In Figure 4.5 there is represented the collection `Teachers` and the members of the collection.

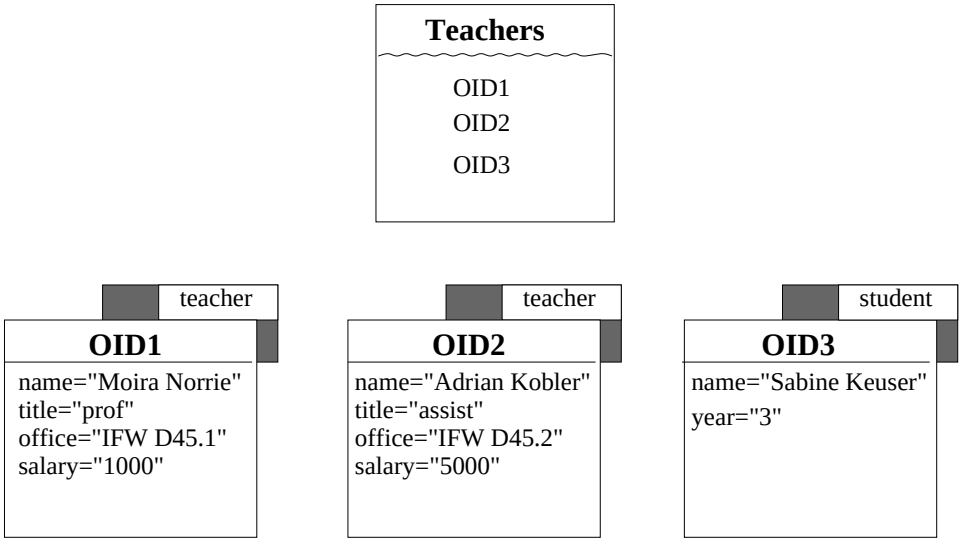

Figure 4.5: Example for illustrating the notion of collection viewed through a type

Objects represented by the identifiers OID1 and OID2 are dressed by the type `teacher` and object given by OID3 is dressed by the type `student`.

So,

```
Teachers{teacher, student}={OID1, OID2, OID3}
Teachers(teacher)={OID1, OID2}
```

```
Teachers(student)={OID3}
Teachers={OID1, OID2, OID3}
```

We will introduce the notion of *base type of a collection*. For a collection we can explicitly specify the base type, according to the semantic of that collection. For instance, for collection `Students` the base type can be specified to be `student`, for collection `Teachers` the type `teacher` and so on. But, if no base type is associated to a collection, it can be established to be the type of the majority of members of the collection.

For a collection $C\{t_1, \ldots, t_n\}$ if the base type is t_i, then it will be denoted by $C[t_i]$. Of course, $C[t_i] = C(t_i)$.

Algebra for collections

For the heterogeneous collections described in the previous section there can be realized the set of operations described in section 2.2.3.

These operations can be realized in two ways:

- **without type checking** meaning that the operations will not take into account the types of the members of the collections. The operations are evaluated in the same way as operations over two sets.

- **with type checking** meaning that the operations will take into account the types of the members of the collections, that means the collections are viewed through a certain type. Note that the operand collections are heterogeneous collections, but the result of the operation will be a homogeneous collection.

When performing an operation with type checking we will take into account the members of the collection dressed by the base type of the collection. Of course, this can be generalized, to take into account for each operand collection only those members instances of a specified type for that collection.

We will choose some of the operations over collections and describe how they are realized without type checking and with type checking.

For all operations we will use the following example. Suppose we have the collection `Students` containing some members as shown in Figure 4.6.

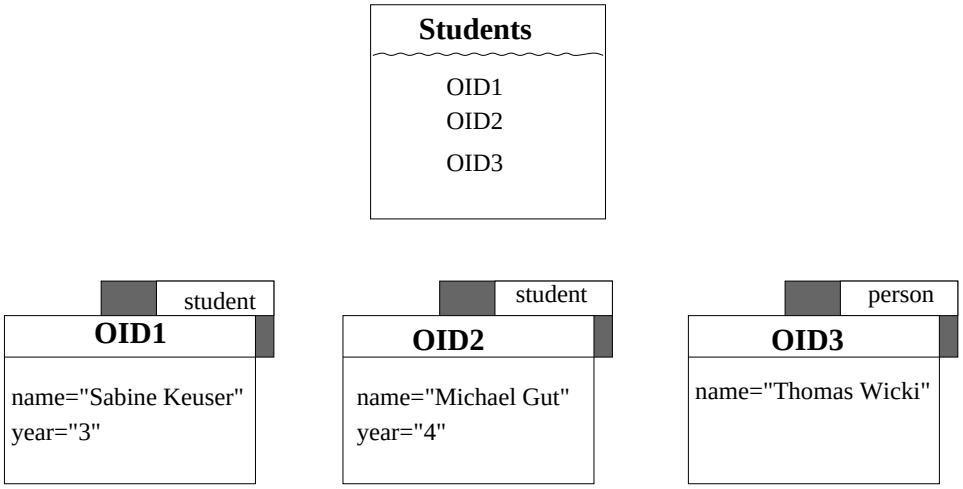

Figure 4.6: Students collection and its members

Also consider the collection `Teachers` with its members, represented in Figure 4.7.

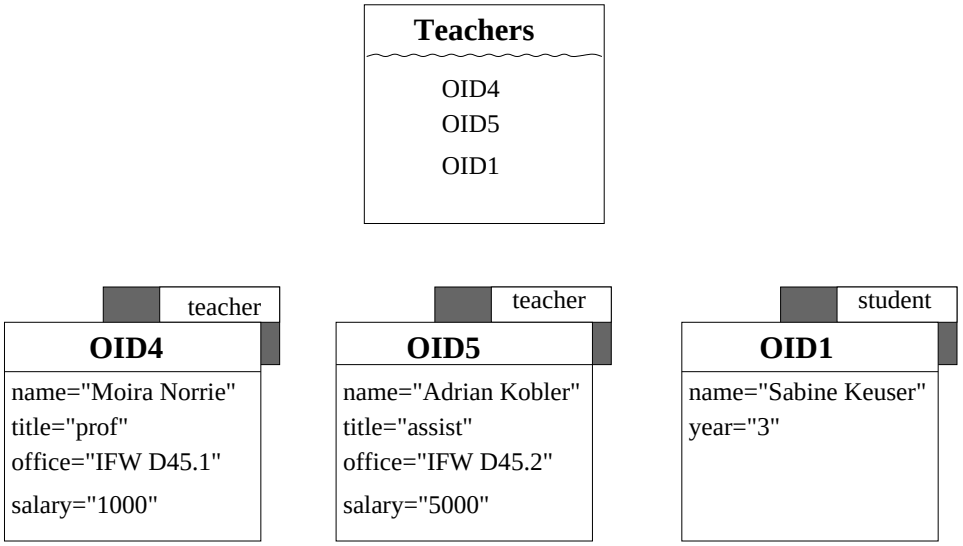

Figure 4.7: Teachers collection and its members

Suppose that the base type of collection `Students` is `student` denoted by `Students[student]`. Also suppose that the base type of collection `Teachers` is `teacher` denoted by `Teachers[teacher]`.

The simplest operation over collections is the **cardinality** operator. For this reason we begin to explain it so that the reader can get the idea of difference between performing an operation without type checking and with type checking.

In case of *cardinality without type checking* we don't take into account the type of the members of the collection. So,

$$\underset{\substack{\text{without}\\\text{type}\\\text{checking}}}{\#} \texttt{Students} = 3,$$

the collection `Students` having 2 members of type `student` and 1 member of type `person`.

In case of *cardinality with type checking* we will count only the members instances of the base type of collection.

$$\underset{\substack{\text{with}\\\text{type}\\\text{checking}}}{\#} \texttt{Students}\ [\texttt{student}] = 2$$

As we said before, there can be done a generalization

$$\underset{\substack{\text{with}\\\text{type}\\\text{checking}}}{\#} \texttt{Students}\ (\texttt{person}) = 1 \quad \text{and} \quad \underset{\substack{\text{with}\\\text{type}\\\text{checking}}}{\#} \texttt{Students}\ (\texttt{student}) = 2$$

Let us consider the **union** operation. When performing union *without type checking* we are not concerned about the type of the members of the collections, so

$$\texttt{Students} \underset{\substack{\text{without}\\\text{type}\\\text{checking}}}{\cup} \texttt{Teachers} = \{\text{OID1, OID2, OID3, OID4, OID5}\}$$

If we denote the result collection `ST_without_T_C`, then it can be graphically represented as depicted in Figure 4.8.

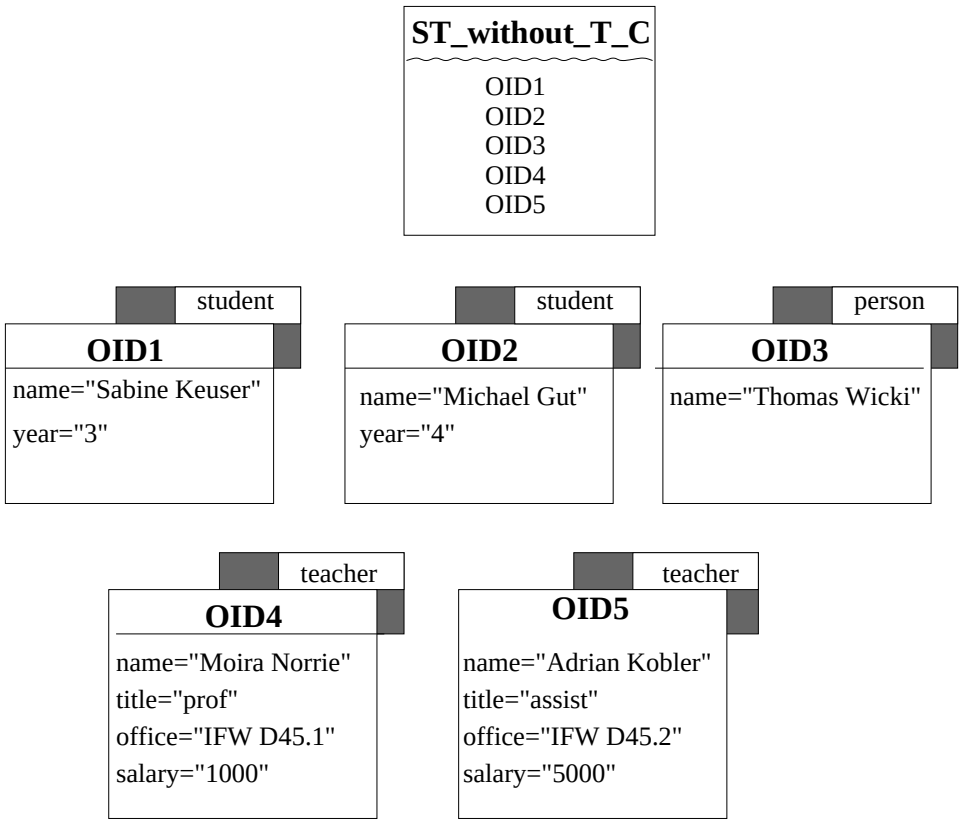

Figure 4.8: Result of union without type checking of collections Students and Teachers

By the following relation we want to show that the types associated to the result operation are `student`, `teacher` and `person`.

$$\texttt{ST\_without\_T\_C}\ \{\texttt{student}, \texttt{teacher}, \texttt{person}\} = \{\ \text{OID1, OID2, OID3, OID4, OID5}\}$$

The result will not have implicitly associated a base type, but it can be established to be the type of the majority of members of the resulting collection. In this case there are 2 members of type `student`, 2 members of type `teacher` and 1 member of type `person`. So, the base type can be either `student` or `teacher`.

In case of *union with type checking* we will consider only the members of the two collections instances of the base types of the collections. The result collection will have as base type the least common supertype of the base types of the collections, that is `student` $\sqcup$ `teacher` (see section 2.2.3). The intersection of properties of the type `student` with the properties of the type `teacher` is the property `name`. So, `student` $\sqcup$ `teacher`=`person`.

Note that a new type would have been created if there is no type having the least common supertype properties.

Let us denote the result collection `ST_with_T_C`. We can write the following mathematical relation:

$$\begin{aligned} \texttt{ST\_with\_T\_C} &= \texttt{Students}\ [\texttt{student}] \underset{\substack{\text{with}\\ \text{type}\\ \text{checking}}}{\cup} \texttt{Teachers}\ [\texttt{teacher}] \\ &= \{\text{OID1, OID2}\} \cup \{\text{OID4, OID5}\} \\ &= \{\text{OID1, OID2, OID4, OID5}\} \end{aligned}$$

`ST_with_T_C` can be graphically represented as shown in Figure 4.9.

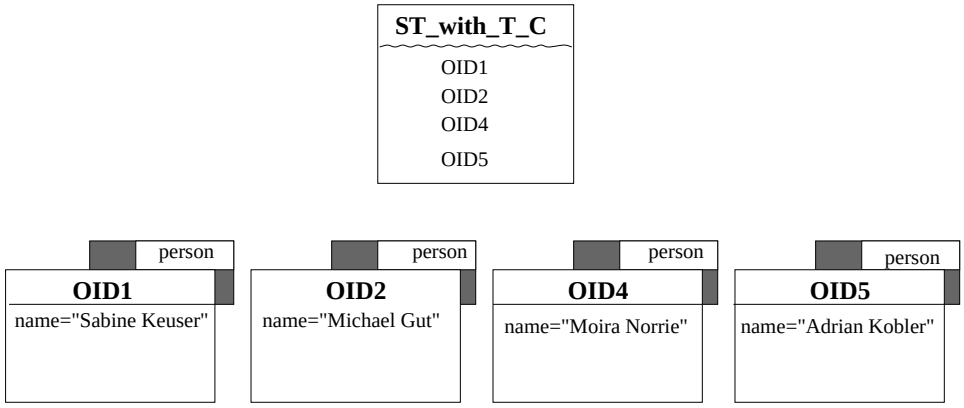

Figure 4.9: Result of union with type checking of collections Students and Teachers

The objects OID1, OID2 have also the attribute `year` and the objects OID3 and OID4 have also the attributes `title`, `office`, `salary`. We consider that an object instance of the base type of the collection can be instance of the least common supertype of the two base types of operand collections.

Another example is the **map** operation. A map operation on a collection applies a function to each element of the collection and forms a collection of the results. We have used a sort of type mapping function, i.e. the function maps each object of the collection to a given type by creating a new object having the properties of that type. For example, consider an object of type `teacher`. We want to map this object to the type `person`. The result object will have only the properties of the type the mapping was done. In our case only `name` attribute is kept, the attributes `title`, `office`, `salary`, being lost. The result of the mapping of object given by OID5 of type `teacher` to the type `person` is shown in Figure 4.10.

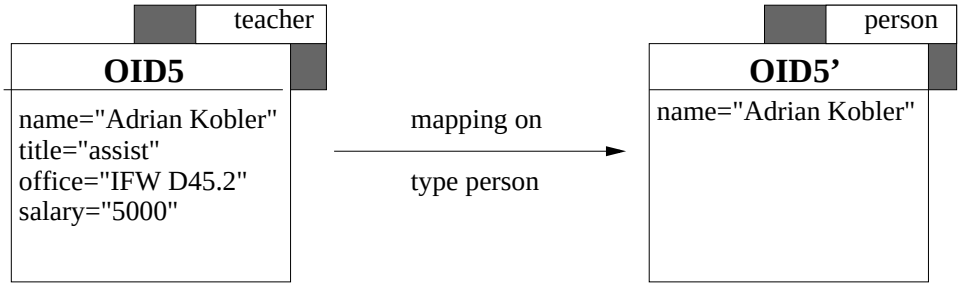

Figure 4.10: Mapping of object OID5 of type `teacher` to the type `person`

We have considered that, when the mapping is done according to one type, the object must have all properties of that type. But, this condition can be relaxed so that the object satisfies the constraint which was set in order to be instance of that type (see section 4.3).

Suppose we want to map `Teachers` collection on type `person`. If we denote the result `MapT_without_T_C`, mathematically this can be written as:

$$\texttt{MapT\_without\_T\_C} = \underset{\substack{\text{without}\\ \text{type}\\ \text{checking}}}{\alpha} \; (\; \texttt{Teachers}\{\texttt{student}, \texttt{teacher}\}, \; \{\texttt{student}, \texttt{teacher}\} \rightarrow \texttt{person})$$

$$= \{\text{OID4', OID5', OID1' }\}$$

Graphically, the result is represented in Figure 4.11.

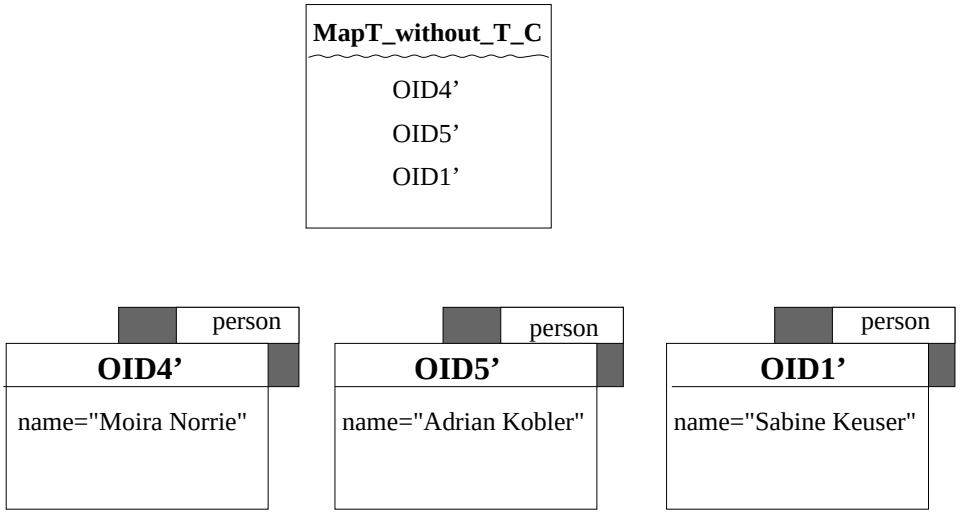

Figure 4.11: Result of mapping collection Teachers on type person

The result collection must have all members instances of the type the mapping was done.

Note that it is possible that two members of the collection to be mapped to the same object in the result collection. For instance, suppose collection `Teachers` contains as members two objects of type `student` having the same value of attribute `name`. That means that there exist two different students having the same name, as shown in Figure 4.12. Then the result would be the same as in the previous case, because OID1 and OID6 are both mapped to OID1'.

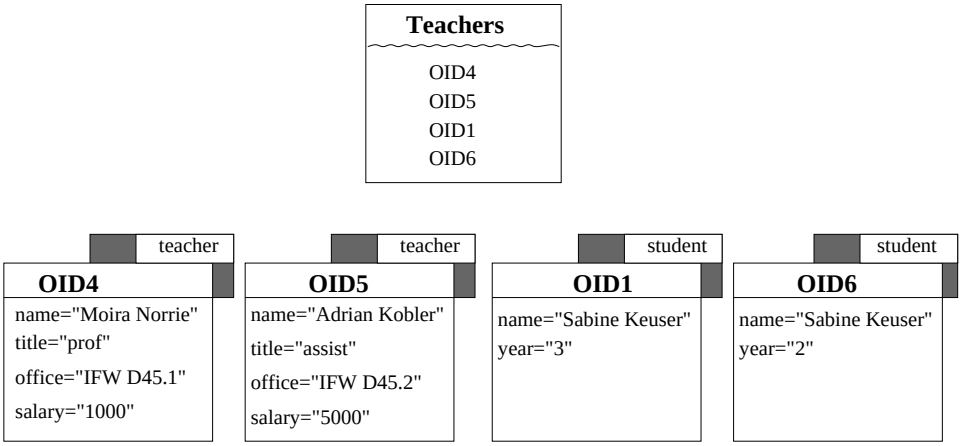

Figure 4.12: Collection Teachers containing two students with the same name

If the mapping is done *with type checking* and we denote the result `MapT_with_T_C`, then we can write:

$$\texttt{MapT\_with\_T\_C} = \underset{\substack{\text{with}\\ \text{type}\\ \text{checking}}}{\alpha} (\texttt{Teachers}\ [\texttt{teacher}],\ \texttt{teacher} \rightarrow \texttt{person})$$

$$= \{\text{OID4', OID5'}\}$$

Here we map only the members instances of type `teacher`.

4.3 Typing

We have seen that in eXtreme Design meta model the set of object attributes and methods is not fixed.

Suppose we create an object and dress it with a certain type. What happens if we remove or add attributes to the object? In case of strong typing, the object would be no more instance of that type. So, typing should allow some flexibility.

We say that an object can be considered to be dressed with a type even if the set of attributes of the object is different than the set of attributes of the type. Of course, this difference cannot exceed some limits.

An object can be dressed with a certain type if one of the following conditions is fulfilled:

1. the object has a **number of common attributes** with the type

2. the object has **at most Δ extra attributes** and **at most Δ missing attributes** compared to the attributes of the type, Δ being specified

3. the ratio between the number of common attributes and the number of different attributes must be greater then a given limit.

$$\frac{\text{no. of common attribs}}{\text{no. of missing attribs} + \text{no. of extra attribs}} \geq \text{percentage},$$

where percentage is given.

Suppose we create an object and dress it with type `student` as shown in Figure 4.13.

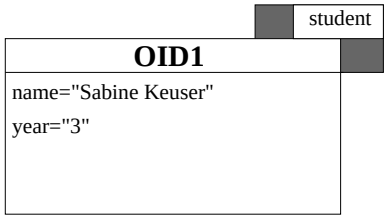

Figure 4.13: The initial state of the object

We will add and remove some attributes and see if the above conditions are fulfilled. Suppose that:

- for condition 1, `no. of common attribs`=2

- for condition 2, Δ=1

- for condition 3, `percentage`=0.5

Suppose we add the attribute `address_of_work` (see Figure 4.14).

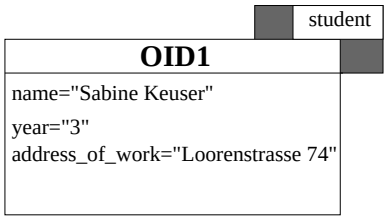

Figure 4.14: After adding the attribute `address_of_work` it is of type `student` according to all conditions

Then,

- condition 1 is fulfilled because `no. of common attribs`=2

- condition 2 is fulfilled because there is only 1 `extra attribute`

- condition 3 is fulfilled because $\frac{2}{0+1} = 2 > 0.5$

Further, suppose we add the attribute `phone_from_work` (see Figure 4.15).

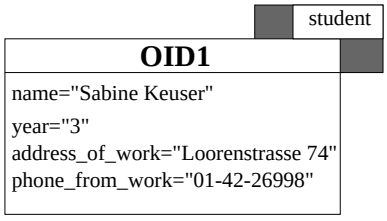

Figure 4.15: After adding the attribute `phone_from_work` it is of type `student` according to first and third conditions

Then,

- condition 1 is fulfilled because `no. of common attribs`=2

- condition 2 is not fulfilled because there are 2 `extra attribs`

- condition 3 is fulfilled because $\frac{2}{0+2} = 1 > 0.5$

Further, suppose we delete the attribute `year` (see Figure 4.16).

OID1
name="Sabine Keuser"
address_of_work="Loorenstrasse 74"
phone_from_work="01-42-26998"

Figure 4.16: After removing the attribute `year` it is no more of type `student` according to all conditions

Then,

- condition 1 is not fulfilled because `no. of common attribs`=1
- condition 2 is not fulfilled because there are 2 `extra attribs`
- condition 3 is not fulfilled because $\frac{1}{1+2} = 0.33 < 0.5$

In this section we have taken into account only the object attributes when deciding if an object is instance of a certain type. But we can consider the methods of the object instead of attributes. The same three conditions can also be applied to methods.

A more general case would be to consider together object attributes and methods. We call the attributes and the methods of an object as being the properties of that object. In this case, the three conditions can be written as follows:

1. the object has a **number of common properties** with the type
2. the object has **at most Δ extra properties** and **at most Δ missing properties** compared to the properties of the type, Δ being specified
3. the ratio between the number of common properties and the number of different properties must be greater then a given limit.

$$\frac{\text{no. of common props}}{\text{no. of missing props} + \text{no. of extra props}} \geq \text{percentage},$$

where percentage is given.

Further, there can be considered that not only the number of properties, but a given set of properties should be taken into account when deciding if an object is instance of a type. For instance, consider type `teacher` having the attributes `name`, `title`, `office` and `salary` and the method `teach_courses()`. Suppose an object has the properties depicted in Figure 4.17. Suppose that according to one of the three conditions, the

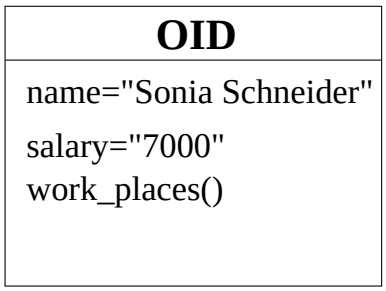

Figure 4.17: An object which satisfies one of the three conditions to be of type teacher, but it's not appropriate to be of this type because of the missing attribute `title` and method `teach_courses()`

object can be instance of type `teacher`. But it's not appropriate to consider this object instance of type `teacher` because of the missing attribute `title` and method `teach_courses()`. We can impose that, in order to be of type `teacher`, an object must have the attributes `name`, `title` and method `teach_courses()`. In this case, object `OID` is not of type `teacher`.

Chapter 5

Implementation considerations

In this chapter we present some implementation issues concerning our framework and the application developed based on this framework. Also, we give an overview of the functions offered by the framework and by the application.

5.1 Implementation issues for the framework

First of all, it must be stated that a *collection* is, in fact, a metaobject, representing an object group. We can insert objects into this object group. The inserted objects represent the members of the collection.

A *type* can be represented as an object to which there are added some properties (attributes and methods). A value of an attribute is a primitive type such as ''`java.lang.String`'' or ''`java.lang.Integer`''. In order to represent that some objects are instances of a type, we have chosen to design the type as a collection whose members are the objects instances of that type. So, a type is, in fact, an object group having some associated properties.

For managing all collections, there is a collection named `Collections`. This collection contains as members all created collections. By iterating on all members of collection `Collections`, we iterate over all collections. A member of collection `Collections` is, at its turn, a collection. Whenever a collection is created, it must be inserted into the collection `Collections`. Let us present a sketch of the function `createCollection` of our framework. This function is used if we want to create a collection with a given name.

```
public String createCollection(String name)
                        throws RemoteException{
   ...
```

```
    // find the identifier of object group "Collections"
    String collsID=metaObject.retrieveOID("Collections");

    // create a new object repesenting the collection
    String oid=metaObject.getNextOID();
    metaObject.createObject(oid, null, name);

    // insert the collection in collection "Collections"
    metaObject.insertMember(collsID, oid);
    ...
    // the function returns the identifier of the object
    return oid;
  }
```

Suppose we want to create the collection `Students`. The XD DML statements for creating the special collection `Collections` and the collection `Students` are the following:

```
//create collection Collections
create object Collections;

//create Collection Students and insert it
//into collection Collections
create object Students;
insert into Collections:[Students];
```

For managing all types and their instances, we have created a collection named `Types`. This collection has as members all existing types, which, as we have seen before, are also collections. Whenever a new type is created, it is inserted in collection `Types`. Let us present a sketch of the function `createType` of our framework. This function is used when we want to create a new type with a given name and a list of attributes. For each attribute there is specified the corresponding domain of values.

```
public String createType(String name, Vector attribs,
                                       Vector values)
 throws RemoteException{
  ...
  // there is created a new object representing the type
  String oid=metaObject.getNextOID();
  metaObject.createObject(oid, null, name);

  // find the identifier of collection Types
```

```
    String typesID=metaObject.retrieveOID("Types");

    // insert the object type into the collection Types
    metaObject.insertMember(typesID, oid);

    // create the attributes associated to the type
    //and assign them the corresponding values
    for(int i=0; i< attribs.size(); i++){
      String attrib=(String)attribs.elementAt(i);
      metaObject.createAttribute(oid, attrib);
      metaObject.assert(oid,attrib, values.elementAt(i));
    }

    ...
    return oid;
  }
```

Suppose we want to create the type `teacher` and assign it the attributes `name`, `title`, `office`, `salary`. For creating the special collection `Types` and the type `teacher`, we can use the following XD DML statements:

```
//create collection Types
create object Types;

//create type teacher and assign it the attributes
create object teacher;
assert(teacher, name, "java.lang.String");
assert(teacher, title, "java.lang.String");
assert(teacher, office, "java.lang.String");
assert(teacher, salary, "java.lang.Integer");

//insert the type object into collection Types
insert into Types:[teacher];
```

Further, for creating an object `teacher1` of type teacher and inserting it into the collection `Teachers`, the following XD DML statements can be specified:

```
//create object teacher1 and dress it
//with type teacher
create object teacher1;
dress teacher1 as teacher values
(
  name = "Adrian Kobler";
  office = "IFW D45.2";
```

```
  title = "assist";
  salary = "5000";
);

//insert teacher1 into the collection
//of instances of type teacher
insert into teacher: [teacher1];
//insert teacher1 into the collection Teachers
insert into Teachers: [teacher1];
//insert teacher1 into the collection Objects
//(collection Objects is explained further
// in this section)
insert into Objects: [teacher1];
```

Further, for making the connection between a collection and the base type associated to the collection, there was implemented a map which is a data structure similar to a hashtable, i.e. we map key to values. Using this map, we can find out for which collection a given type is the base type and conversely, for a collection which are the base types associated to it. As we presented in section 4.2, usually for a collection there is associated a base type. But, maybe the user wants to associate more base types to a collection. Suppose we have a collection called `University_Members` containing objects of type `student` and `teacher`. It would be appropriate to create the new type `person` and assign it to be the base type of the collection `University_Members`. But, suppose the user does not want to create a new type for this collection. Also, if there is not associated a base type to the collection, the base type would be the type of the majority of members of the collection. Suppose the base type would be established to be `student`. This is not semantically appropriate, because collection `University_Members` contains not only students but also teachers. So, the most convenient solution to this problem would be to associate two base types to the collection, i.e. the types `student` and `teacher`.

The map was implemented to be a collection containing as members objects with associated attributes `type` and `coll`. The meaning of these attributes is the following: `type` represents a type given by its identifier and `coll` represents the list of collections given by their identifiers, for which `type` is their base type. The list of collections associated to a type was easily implemented by using multiple attributes.

For example, for creating the map and for expressing that the base type of collection `Students` is the type `student`, we can use the following XD DML statements:

```
//create the hashtable collection
create object TypeCollHashtable;
```

```
//Create type typeCollRel.
//All members of TypeCollHashtable
//collection must be dressed with
//this type.
create object typeCollRel;
assert(typeCollRel, type, "java.lang.String");
assert(typeCollRel, coll, "java.lang.String");

//create the object containing the relation between
//type "student" and collection "Students"
//and insert it into TypeCollHashtable collection
create object studStudRel;
dress studStudRel as typeCollRel values
(
  type=student;
  coll=Students;
);

insert into TypeCollHashtable:[studStudRel];
```

Note that the previous DML statements assign the name of the type and the name of the collection, respectively, to the `type` and `coll` attributes of object `studStudRel`. The framework will deal with the type identifier and collections identifiers. An application which uses our framework and loads the DML file containing the previous statements, must perform the conversion from the names to the identifiers. We motivate that it is better to deal with object identifiers and not with their names by the fact that an identifier is unique. And it also can be the case that there is no name assigned to the object.

Also, there was implemented a collection `Objects` containing all created objects. This collection is useful when we want to iterate over all existing objects.

Note that in our framework it is not possible to create two collections having the same name. The same rule applies to the types. An exception is thrown in the functions for creating a collection or a type if the name passed as parameter is an already existing name for a collection or a type.

For implementing the type constraint to be used when deciding if an object is anymore instance of a type (see section 4.3), it was created an object with alias `setting`. By default, it is used the percentage constraint with a value of 50. In this case, it is added the attribute `percentage` to the object `setting` and it is assigned value 50 to the attribute `percentage`. In case the user sets another constraint, the attribute `percentage` of the object `setting` is removed. If a Δ constraint is defined, then there is added the attribute

`delta` having a given value. If there is a common number of attributes constraint defined, then there is added the attribute `noCommAttrib` having a given value. So, every time a constraint is defined, the old attribute of the object is removed and there is added the attribute corresponding to the new constraint. By modelling the typing constraints in this way, one can easily create new constraints by simply adding new attributes to the object `setting` and implementing a `dress` function for each new type constraint.

For implementing binary collections, it was created a type we call `association` which, essentially, is the representation of a relation between objects. In our case, this is an object having the attributes `source` and `target`. Every member of a binary collection must be dressed with the type `association`. The attribute `source` of a member of the binary collection will have as value the object identifier for the source object of the association. The attribute `target` will have as value the object identifier for the target object of the association.

The subcollection relationship between two collections was implemented by constructing a relation between these two collections.

5.2 Functionality of the framework

5.2.1 Classifying

Concerning the classification layer, our framework offers functions for the following operations:

- creation and update of a collection
  - create a collection with a given name
  - check if an object identifier or name represents a collection
  - insert an object into a collection
  - check if an object is member of a collection
  - remove an object from a collection
  - find all collections a given object is member of
  - display a given collection
  - find all members of a collection and find the collection viewed through a certain type
- subcollection management
  - create a subcollection relationship between two collections
  - check if two collections are in subcollection relationship
- special collections and their management(Objects, Collections, Types, TypeCollHashtable)

  - create a relationship between a collection and a type, representing the fact that the type is the base type of the collection
  - from the hashtable, find the collections for which a given type is the base type
  - from the hashtable, find the types associated to a collection

- find the base type of a given collection, in case there must be only one base type associated to a collection
- find all types, either associated to the collection in the hashtable, or being the types the collection members are dressed with
- algebra for collections
  - cardinality with and without type checking of a collection
  - union with and without type checking of two collections
  - intersection with and without type checking of two collections
  - difference with and without type checking of two collections
  - selection with and without type checking applied on a collection. There was implemented the equal selection operator, i.e. to select all members of a collection satisfying the property that a given attribute has a given value. But, there can be easily realised some other selection operators by implementing the `SelOp` interface.
  - map with and without type checking. The mapping operation is done according to a given type.
  - reduction operator with and without type checking. There were implemented the sum, average and maximum reduction operators. That is, there can be found, respectively, the sum, average or maximum of values of a given attribute of the members of the collection. There can be easily realised some other reduction operators by implementing the `RedOp` interface.
- binary collections
  - create an object member of a binary collection, showing the association between two objects
  - find the source object corresponding to a given target object in a binary collection
  - find the target object corresponding to a given source object in a binary collection
  - find the domain of a given binary collection
  - find the range of a given binary collection

5.2.2 Typing

Concerning typing, the functions of the framework implement the following operations:

- create a type with a given name and given attributes together with the domain of values for these attributes
- find if there exists a type having a given list of attributes
- find if an object is instance of a given type
- find all types an object is instance of
- set a constraint when deciding if an object is instance of a type (see section 4.3)
  - common number of attributes constraint
  - Δ constraint
  - percentage constraint
- find if an object is of a given type according to one of the following constraints:
  - common number of attributes constraint
  - Δ constraint
  - percentage constraint
- dress an object with a given type if one of the following constraints is fulfilled:
  - common number of attributes constraint
  - Δ constraint
  - percentage constraint
- find the types an object should be stripped, i.e. the types an object is no more instance of according to the constraint which was set
- strip an object from a certain type
- find the greatest common subtype of two types
- find the least common supertype of two types
- check if two objects have equal attributes
- display all existing types

5.2.3 Generation of schema definitions

Also, the framework offers some functions in which information is analysed and there are generated schema definitions which can be used by OMS Java, i.e. DDL (Data Definition Language) and DML (Data Manipulation Language) files.

In the generated DDL file there are listed all existing types together with their attributes. For example, for type `student` there are written the following OMS Java DML statements:

```
type student
(
      year :java.lang.Integer;
      name :java.lang.String;
);
```

Also, the generated DDL file contains the list of existing collections. Each collection has associated all the types its members are instance of. The list of types associated to a collection always begins with the base type of the collection. Let us consider that collection `Students` contains members which are instances of type `student`, and collection `Teachers` contains members which are instances of type `teacher`. Suppose we perform the union without type checking between collections `Students` and `Teachers`. Let us name the result collection `Persons`. Because the union was performed without type checking, there is not associated a base type to the collection `Persons`. In this case, the base type will be established to be the type of the majority of members of the collection. Suppose there are more students than teachers, so the base type of collection `Persons` is type `student`. Then, the generated DDL statements for the collections `Students`, `Teachers` and `Persons` are the following:

```
collection Students: set of student;
collection Teachers: set of teacher;
collection Persons:  set of (student, teacher);
```

The last DDL statement it is not yet supported by OMS Java, because in case of the OM model, a collection is homogeneous, i.e. it has associated only one type. A solution to this problem would be to generate the following DDL statements:

```
collection Persons1: set of student;
collection Persons2: set of teacher;
```

But in this case all information concerning collection `Persons` should be reanalysed. For example, in the DML file, the statements for inserting an object into collection `Persons` should be modified. If the object is of type

`student`, then it should be inserted into collection `Persons1`. If the object is of type `teacher`, then it should be inserted into collection `Persons2`. In our example, another solution would be to analyse the percentage of members of type `student` and of type `teacher`. We said that there are more students than teachers. If, for example, there are 98% objects of type `student` and 2% objects of type `teacher`, then the solution is to associate type `student` to the collection `Persons`. So, the following DDL statement is generated:

```
collection Persons:  set of student;
```

But if there are 51% objects of type `student` and 49% objects of type `teacher`, then a solution is to associate the least common supertype of types `student teacher` to the collection `Persons`. We know that `student` $\sqcup$ `teacher=person`. So, the following DDL statement is generated:

```
collection Persons:  set of person;
```

Also, the generated DDL file contains a list of constraints. Because in our framework it was implemented only the subcollection constraint, only this constraint will be generated.

For instance, consider that collections `Students` and `Teachers` are subcollection of collection `Persons`. This can be expressed by the following OMS Java DML statements:

```
constraint Students subcollection of Persons
constraint Teachers subcollection of Persons
```

In the DML file, for each object there are specified the types the object is dressed with. Also, there are specified the values of the attributes of the object according to each type the object is dressed with. Also, for each object there are specified the collections it is member of. For example, consider an object `teacher1` having the attributes as shown in Figure 5.2.3.

teacher1
name="Adrian Kobler" title="assist" office="IFW D47.1" phone="41-1-63-27244" email="kobler@inf.ethz.ch"

Figure 5.1: An object dressed with types `teacher` and `person`

Suppose that the object was dressed with type `teacher` and type `person`. Type `teacher` has the attributes `name`, `title`, `office` and `salary` and type

person has the attribute name. The following OMS Java DML statements show that the object teacher1 is instance of types teacher and person and there are specified the attributes of object teacher1 according to these types.

```
create object teacher1;
dress teacher1 as teacher values
(
    name="Adrian Kobler";
    title="assist";
    office="IFW 45.2";
);

dress teacher1 as person values
(
    name="Adrian Kobler";
);
```

There are some attributes of object teacher1 which are not included in the set of attributes of the types the object is dressed with. For showing all attributes of an object in the DML file, there is created a temporary object representing a type. This temporary object has as attributes all the object attributes which do not belong to any of the types the object is dressed with. For example, in our case, there are generated also the following OMS Java DDL statements:

```
type tempXD_67
(
   phone: java.lang.String;
   email: java.lang.String;
);
```

and the following OMS Java DML statements:

```
dress teacher1 as tempXD_67 values
(
    phone="41-1-63-27244";
    email="kobler@inf.ethz.ch";
);
```

For simplicity, we specify that the attributes of these temporary types are of type "java.lang.String". For specifying the appropriate types for the attributes, information should be reanalysed.

As we said before, for each object in the DML file, there are specified the collections it is member of. Suppose that object teacher1 belongs to the collections Teachers and Persons. The following DML statements insert object teacher1 into the corresponding collections:

```
insert into collection Teachers: [teacher1];
insert into collection Persons: [teacher1];
```

Also, the DML file contains all relationships between objects. Suppose object `teacher1` represents a teacher and objects `course1` and `course2` represent two courses `teacher1` teaches. The following DML statements create the relationships between `teacher1` and the courses `course1` and `course2`:

```
association(teacher1,course1);
association(teacher1,course2);
```

5.3 Functionality of the Application

In this section we present a "web application" which can be considered as prototype for RIM.

The application offers the menu depicted in Figure 5.2.

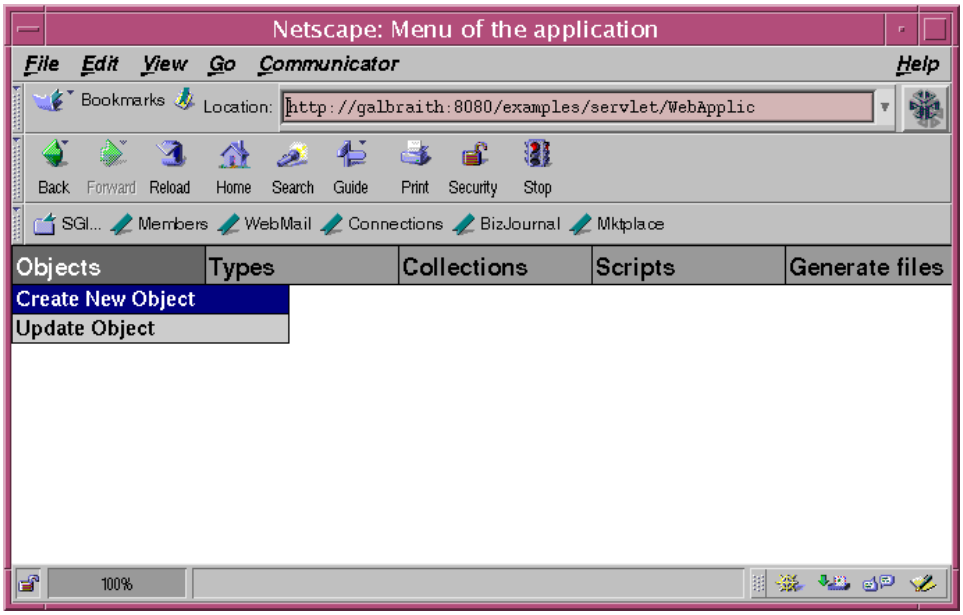

Figure 5.2: The menu of the application

Objects menu provides two submenus: **Create New Object** and **Update Object**.

Create New Object allows the user to create a new object. First, the user has the possibility to choose between one of the existing types. If there is selected one of the existing types, the new object will be dressed with this

type, so it will have all the properties of the type. Further, the user can enter values for these attributes, can add new or remove object attributes. When there is added or removed an attribute, there is checked if the condition for dressing the object with the choosen type is fulfilled. If the condition is no more valid after adding or removing an attribute, there is sent a message to the user. So, if the user wants to keep the object instance of that type, he can undo the last operation. After updating the object attributes, the user is asked to insert the object into a collection. The user can choose among one of the existing collections or he/she has the possibility to create a new collection in which to insert the object. If the user does not know in which collection to insert the object, he/she has the possibility `''Don't know''`. Using this option, the user will find the name of the collections the object is most appropriate to be inserted into. These collections are the ones for which the type the object is dressed with is their base type. If the user does not want to choose a type from the list of existing types, he/she can choose the option `''None of the above''`. Using this option, the user can also create a new type having the same attributes as the created object.

Update Object will update the properties of an object. To select one of the existing objects, the user can enter known information about that object in the form *attribute name=attribute value*. After this, there will be displayed a list of objects which met the user criteria. For each object, we list all attributes and their values, so that the user can choose the object he/she is searching for. For the selected object, we display the names of the attributes together with their values and the list of objects the choosen object is in relation with. The user can change the values of the existing attributes, add or remove attributes, add new relations. For choosing a new object to be added to the list of relations of the object, we have provided the same steps as for choosing the old object. Suppose we want to update an object of type `teacher` for which we know only that the `name` is *Adrian Kobler*. We have provided this information and we have found the object we wanted. The object has the following properties: `name`=*Adrian Kobler*, `title`= *assist*, `office`=*IFW D45.2* and `salary`=*5000* and also it has a relation with an object of type `course`. In the figure Figure 5.3 it is represented the state of the object after removing the attribute `office` and modifying the value of attribute `salary`.

The menu **Types** has the two submenus **Set Constraint** and **Display Types**.

For **Set Constraint** the user can choose one of the three conditions: number of common attributes constraint, delta constraint, percentage constraint (see section 4.3). When one of the conditions is selected, the corresponding value can be entered. In Figure 5.4 it is shown that it was choosen the percentage constraint and the value of the percentage was set to be 50.

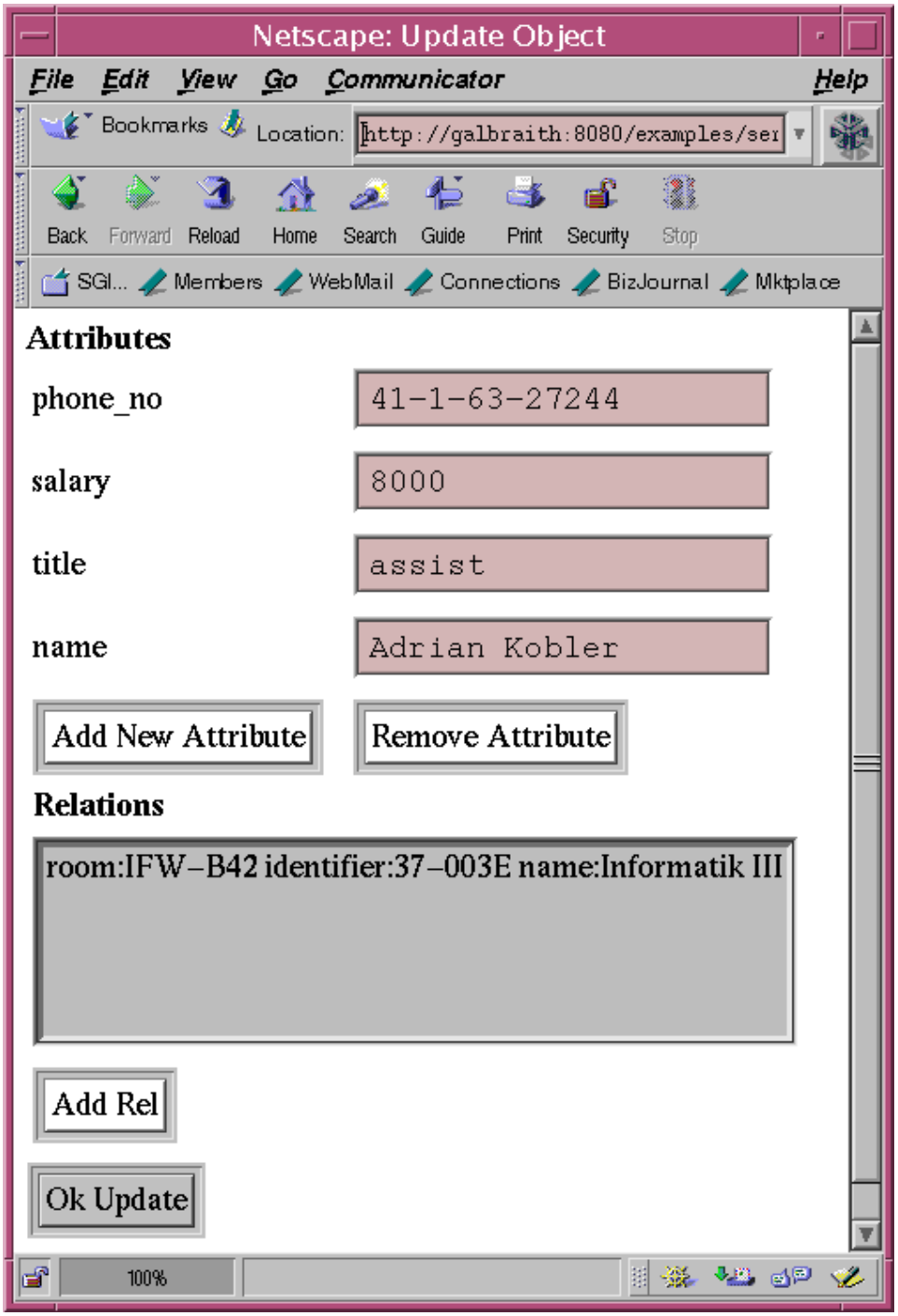

Figure 5.3: The update of an object

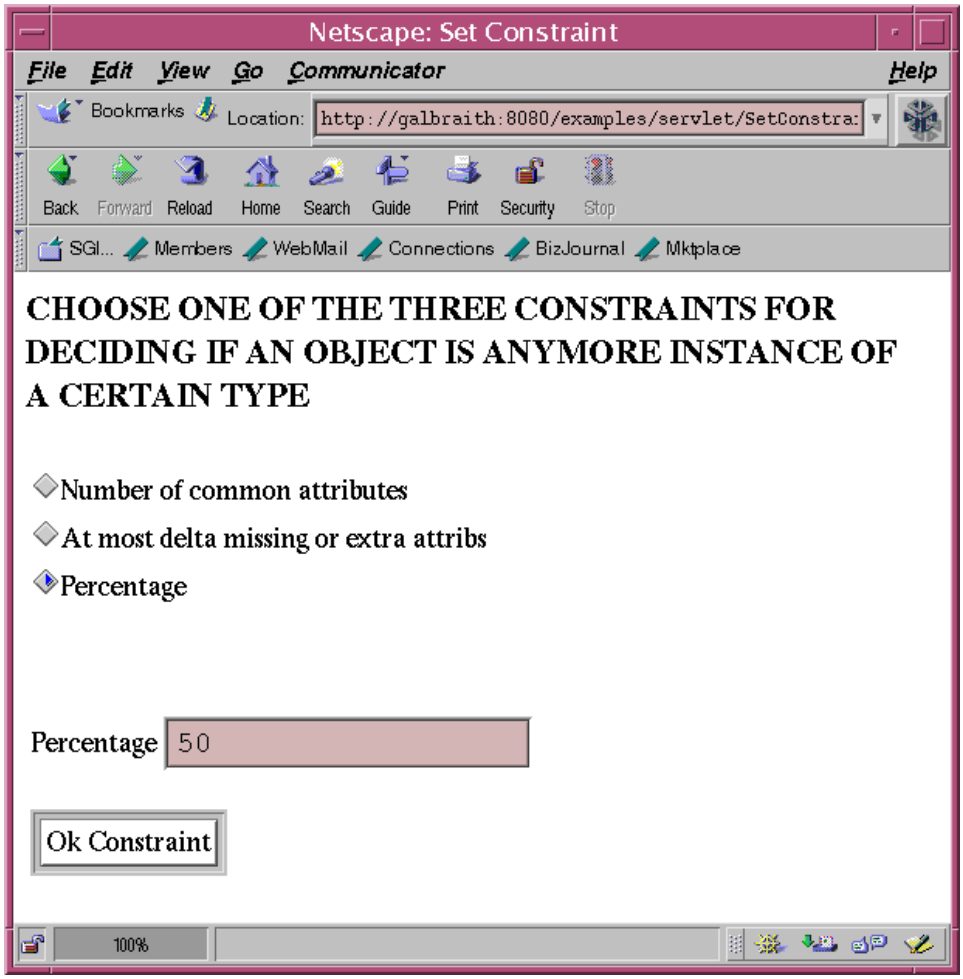

Figure 5.4: The setting of the constraint

Display Types will display all existing types. For each type there will be displayed the attributes and all objects instances of that type.

The menu **Collections** provides two submenus: **Display Collections** and **Update Collection**.

Display Collections will display all existing collections. For each collection there are displayed the attributes of its base type, the members of the collection with their attributes and relations with other objects. There are also displayed the special collections: `Objects`, `Types` and `TypeCollHashtable`. In Figure 5.5 it is shown the collection `Courses`.

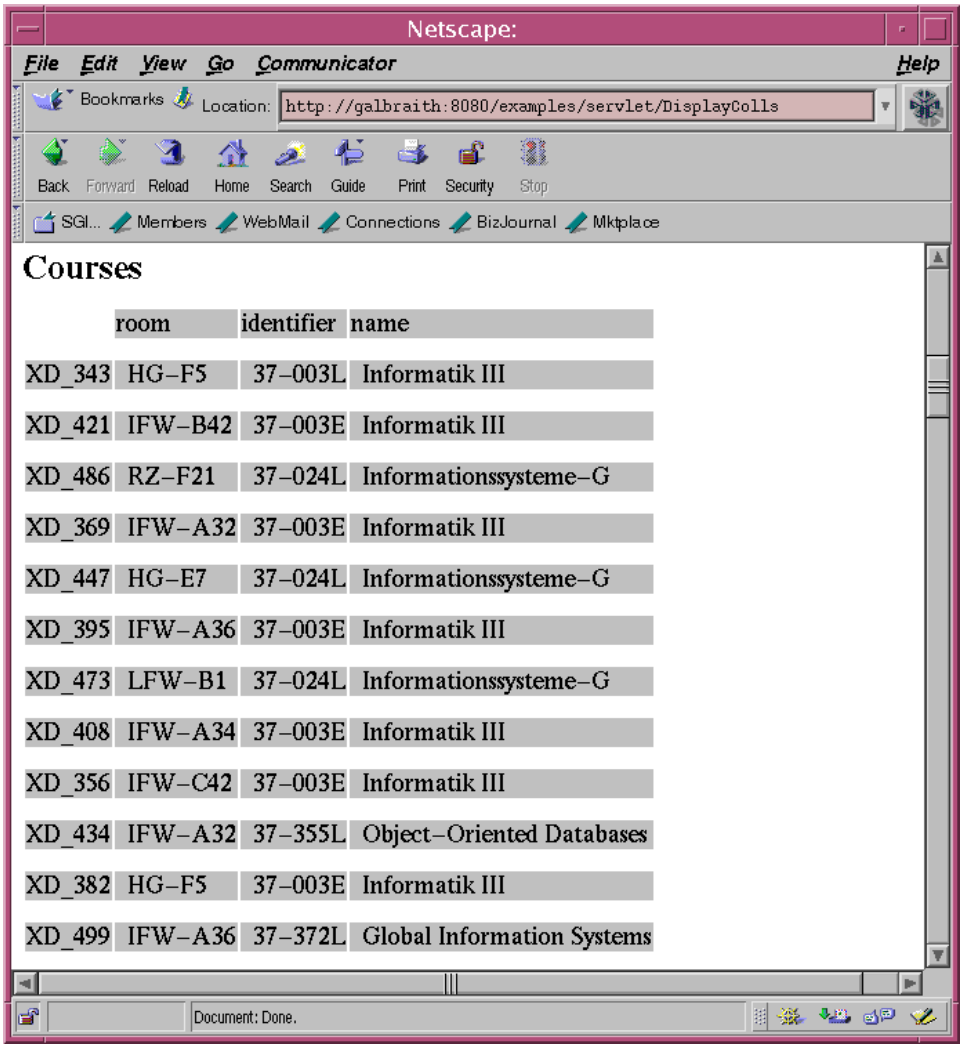

Figure 5.5: The display of collections

Update Collection allows the user to choose a collection, add and remove objects from it.

The menu **Scripts** offers two submenus: **Query without selection** and **Query with selection**. The menu **Scripts** offers algebra operations for collections. There can be performed operations on collections such as union, intersection, difference, selection, mapping, reduction. If a user knows the syntax of AQL (Algebraic Query Language), he/she can use the submenu **Query without selection**. A user who has no knowledge about AQL can use the submenu **Query with selection** and he/she will be guided to select

the parameters necessary for performing different operations.

By choosing the submenu **Query without selection**, the user can type a query in AQL and can choose to perform the operation either with type checking or without type checking. In what follows we will give some examples of AQL queries.

- The AQL query for performing the union between `Students` and `Teachers` is: `Students` **union** `Teachers`.

- The AQL query for selecting all objects members of collection `Teachers` having the value of attribute `title` equal with the string *assist* is: **all** `P` **in** `Teachers` **having (** `P@title`=*assist*)

- The AQL query for obtaining a collection with the `name`s of all students in collection `Students` is: **map** `P` **in** `Students` **by** `P@name`

- The AQL query for finding the sum of `salary` values of all teachers in collection `Teachers` is: **reduce** `P` **in** `Teachers` **aggr** `Q` **by** `Q+(P@salary)` **default** *0*

  For submenu **Query without selection** one can perform only the sum of values of an attribute on a given collection.

If the syntax of the query is not valid it will be generated an error message. If the syntax of the query is valid, the user will be asked to provide, if necessary, the name of the result collection and the name of the result type.

When using **Query with selection**, one can choose an operation from a list of operations and specify if the operation is performed with or without type checking. Further, the user will be asked for the parameters necessary for the choosen operation. For example, suppose it was choosen the mapping operation with type checking. Note that the map function is a type mapping function. In Figure 5.6, for example, it is shown that it will be performed the mapping of collection `Teachers`. It was choosen to perform the mapping not on an existing type, but on one of the attributes of the base type of collection `Teachers`. The attribute on which the mapping is performed was selected to be `name`. It was specified that the name of the result collection will be *TeacherNames* and the name of the type will be *teacherName*.

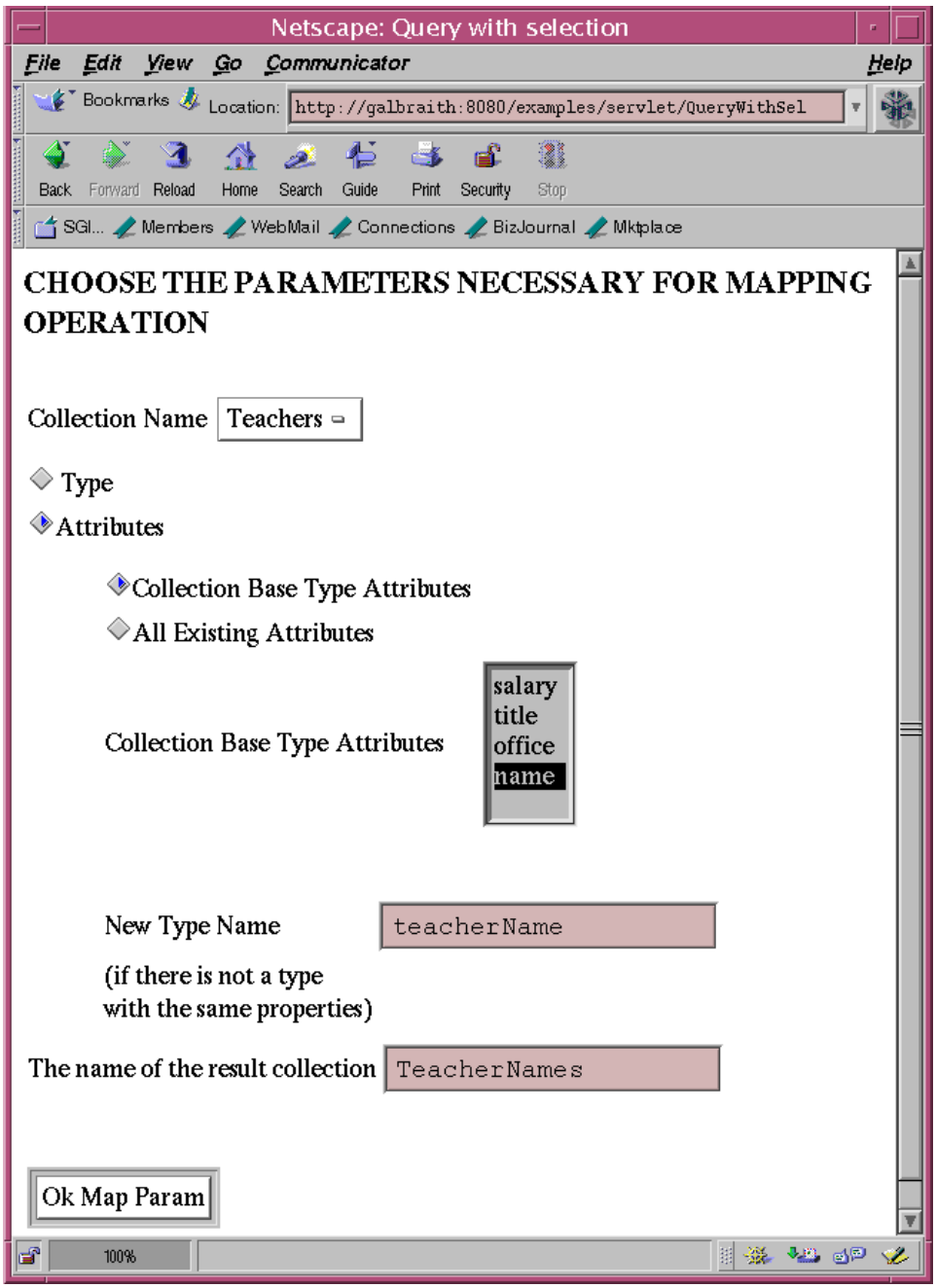

Figure 5.6: The mapping operation

The menu **Generate files** has the two submenus **DDL file** for which there is displayed the content of the DDL file and **DML file** for which there is displayed the content of the DML file (see section 5.2.3).

5.4 Implementation issues for the application

For the connection between the browser and our application we have used Servlets [Mos99]. Why did we choose servlets and not something else? In the following we give a brief description of servlets, emphasizing their advantages.

A Servlet is a Java program(class) that runs on a Web server and is similar to a conventional CGI program. The advantages of Servlets are:

- Servlets are more efficient than CGI processes

  When a CGI process is executed, a new process must be started. Once the program has executed a request, it is terminated. This results in high overhead and can appreciably slow down servers. A servlet is kept in memory, i.e. once it is started, it remains in memory and, therefore, can be used to fulfill multiple requests.

- Servlets are save

  They run in the same address space as the server. Thus, the server is potentially vulnerable to attack by some Servlets. But, Java has a protection mechanism that prevents such attacks, thereby making it safe to run in the server's address space.

- Servlets are written in Java

  This makes it easy to port it to different platforms and Web servers.

For allowing the application to be easily extended, the work was divided on small tasks and each task was realised by implementing a servlet. So we have a servlet for each of the following tasks:

- for finding an object
- for choosing an object
- for adding a new attribute to an object
- for removing one of the attributes of the object
- for creating a new object
- for updating an object
- for choosing a collection
- for updating a collection
- for setting one of the three constraints (see section 4.3)
- for displaying all collections

- for displaying all types
- for displaying the content of the DDL file
- for displaying the content of the DML file
- for allowing the user to introduce a query in AQL for performing operations on collections with and without type checking
- for guiding the user in selecting the parameters for the operations on collections with and without type checking
- for realising the menu of the application

These servlets communicate with each other. The problem that appeared was that these servlets needed to share the workspace, they needed to have access to all meta objects. So, it was implemented a class called `XDWorkspaceManager` which provides functions for getting and setting the meta objects. This class keeps an updated version of the workspace and it does not permit to be garbage collected when it is no more refered. For initializing the workspace only once, `XDWorkspaceManager` contains a static booolean variable which is set to be true when the workspace is created first. When a servlet wants to get meta objects, it checks if the workspace was already initialized by testing this boolean variable. If the workspace was not already created, it calls the static function of this class for performing the initialization. For not allowing the garbage collection, it is used a static variable for permanently keeping the reference to the class. A sketch of this class is provided in what follows:

```
public class XDWorkspaceManager{
   //for not allowing garbage collection
   static XDWorkspaceManager instance=new XDWorkspaceManager();

   //reference to XD workspace which
   //manages all meta objects
   static XDMetaObject metaObject;

   //used to check if it was performed
   //the initialization
   static boolean created=false;

   // for creating the workspace
   static void createRAM(){
       ...
   }
```

```
    // for importing the DML file
    static void importDML(){
        ...
    }

    public static void init(){
      createRAM();
      importDML();
      created=true;
      ...
    }

    public static XDMetaObject getMetaObject(){
      return metaObject;
    }

    public static void setMetaObject(XDMetaObject m){
      metaObject=m;
    }
}
```

5.5 Considerations of installation

In this section we present the structure of our package and describe the steps for installing the application.

We have created a package called `myXDpackage` containing the following files:

- `Colls.java` in which we have defined all functions for the classification activity
- `Types.java` in which we have defined all functions for the typing activity
- `SelOp.java` defining the interface for the selection operator
- `Equal.java` containing the implementation of equal selection operator
- `MapOp.java` defining the interface for the mapping operator
- `MapToType.java` containing the implementation of type mapping operator
- `ReduceOp.java` defining the interface for the reduction operator
- `SumOfField.java` containing the implementation of sum reduction operator

- `AvgOfField.java` containing the implementation of average reduction operator
- `MaxOfField.java` containing the implementation of maximum reduction operator
- `QueryParser.java` containing the implementation of a query parser.

An application which wants to use our framework, must import the `myXDpackage` package.

We have run the application using Java Server Web Development Kit (JSWDK) 1.0.1 and Java Development Kit (JDK) 1.2.

One can choose to create a new web application or to install the application files into the `examples` directory of JSWDK. By web application we understand a collection of resources that is mapped to a specific Uniform Resource Identifier (URI) prefix. The resources include, in our case, Servlets and HTML files. In what follows, we will describe the steps for creating a new web application:

1. There must be created a directory for the web application. Let us call it WEBAPP. The directory should have the same structure as `examples` directory, in our case only subdirectories `servlets` and `WEB-INF`. One way to do this, is to copy the `examples` directory and edit the files.
2. The new application must be added to the JSWDK by editing the file `webserver.xml` in the root JSWDK directory. For example, to create a WEBAPP application, there must be done, at the appropriate location, the following additions to the file:

```
<WebApplication id="WEBAPP" mapping="/WEBAPP"
                            docBase="WEBAPP"/>
```

3. Add the following files to the WEBAPP/WEB-INF/servlets directory:
   - `ChooseCollection.java`
   - `ChooseObj.java`
   - `CreateNewObject.java`
   - `DDL.java`
   - `DML.java`
   - `DisplayColls.java`
   - `DisplayTypes.java`
   - `FindObject.java`
   - `NewAttribute.java`
   - `NewCollection.java`

- `QueryWithSel.java`
- `QueryWithoutSel.java`
- `RemAttribute.java`
- `SetConstraint.java`
- `UpdateColl.java`
- `UpdateObject.java`
- `WebApplic.java`
- `XDWorkspaceManager.java`

4. Compile the files.

5. The server must be restarted.

6. The file `WebApplic.html` must be copied in the `webpages` subdirectory of JSWDK directory. The parameter `action` of the form contained in this file must be updated, i.e. the web application URI name must be changed. In our example, the action parameter will be http://localhost:8080/`WEBAPP`/servlet/WebApplic

7. The file `MyDML.dml` must be copied in the root directory of JSWDK. This file contains the DML statements in case of University example. It must be parsed when the workspace is initialized, i.e. when, for the first time, a servlet wants to get meta objects. After starting the server, the current directory is the root directory of JSWDK. The file `MyDML.dml` must be placed in the root directory of JSWDK in order to be found after the server was started. Another possibility is to change the CLASSPATH so that file `MyDML.dml` can be found.

A simpler possibility, as we have already mentioned, would be to skip steps 1 and 2 and to copy all files mentioned in step 3 in the subdirectory examples/WEB-INF/servlets. At step 6, the action parameter will be http://localhost:8080/`examples`/servlet/WebApplic.

In the browser (we have used Netscape) there must be specified the URL http://localhost:8080/WebApplic.html which will call `WebApplic` servlet.

5.6 Open issues

In this section we will report some open issues and we will give some solutions for them.

When there is performed the union or intersection of two collections, there must be computed the least common supertype, respectively, the greatest common subtype. We have considered that an object which is instance of the base type of one of the collections can be instance of the least common

supertype and of the greatest common subtype of the two base types of the collections.

Let us consider the `Students` collection in Figure 4.6 and the `Teachers` collection in Figure 4.7. Suppose we perform the union with type checking of collections `Students` and `Teachers`. The result of this union operation is depicted in Figure 4.9. Suppose we have choosen the Δ constraint with Δ=2. In collection `Teachers` consider the object OID1 of type `teacher`, having the attributes `name`, `title`, `office` and `salary`. The same object OID1 appears in the result of union with type checking of the two collections. So, the object OID1 is dressed with type `person`. Type `person` is the least common supertype of type `teacher` and of type `student`. But, according to Δ constraint, OID1 cannot be of type `person` because it has 3 extra attributes: `title`, `office` and `salary`, compared to the attributes of type `person`.

When performing the union or intersection with type checking there should be checked if the objects in the result collection satisfy the constraint condition, i.e. if they can be dressed with the base type of the result collection. If an object in one collection which should be inserted in the result collection does not satisfy the constraint condition, then it shouldn't be inserted in the result collection.

Also, when performing the type mapping of a collection on a given type, we have considered that all objects in the result collection must have all the properties of the type the mapping was done. Let us consider the example depicted in Figure 5.7. Suppose the type `teacher` has the following attributes: `name`, `title`, `office` and `salary`. Suppose the type `employee` has the following attributes: `name`, `birthdate` and `salary`. Also, suppose we consider Δ constraint with Δ=2.

We have considered that the object OID5 cannot be mapped to type `employee` because it does not have the attribute `birthdate`. But according to Δ constraint this mapping could be possible.

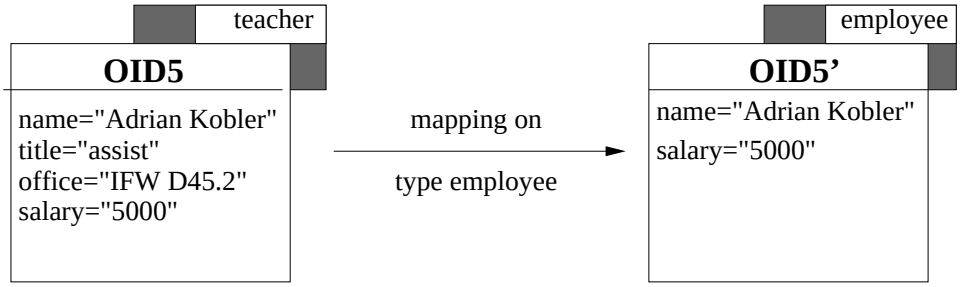

Figure 5.7: The mapping of object OID5 of type `teacher` to the type `employee` should be possible according to Δ=2 constraint

So, the condition for performing the mapping of an object on a certain type can be relaxed so that the object is allowed to have not all the attributes of the type the mapping is done, but the constraint which was set has to be fulfilled.

To summarize, we have to modify the functions for performing the union and intersection with type checking and also the type mapping functions, to check if the objects in the result collection satisfy the type constraint condition.

Chapter 6

Future Work and Conclusions

In this chapter we present some future work issues and conclusions on the project.

6.1 Future Work

In what follows we present some issues for extending our framework.

- After the schema was generated there can be created a *hierarchy of types*. This hierarchy can be created by analysing not only the attributes of the types, but also information concerning objects and collections.

  For example, consider the types `teacher`, `student` and `person`. Suppose, type `teacher` has the attributes `name`, `title`, `office` and `salary`. Also, suppose type `student` has the attributes `name` and `year` and type `person` has the attribute `name`. According to the types attributes, we can infer that type `person` is the supertype of types `teacher` and `student`. But taking into account only the attributes, or even the properties of a type, can lead to mistakes. For example, we can erroneously infer that the type `course`, having the properties `name`, `identifier` and `room`, is subtype of type `person`. Types `course` and `person` are not semantically related. The two types represent totally different concepts, so there does not exist any subtype relationship between them.

  After taking into account the attributes of the types and find a possible type-subtype relationship between the two types, we can check if there exist some objects instances of both types. If there exist such objects, it means that the two types are semantically related. In our example, it does not make sense to have an object instance of type `person` and

of type `course`. But, it can be the case to find objects dressed with type `person` and with type `teacher` or `student`.

When establishing a subtype relationship between two types, it is useful to check if there exist collections containing members of both types. If there exist, it means that the two types are semantically related, because a collection means grouping objects having some common features. In our example, it does not make sense to find in the same collection objects of type `person` and of type `course`. But, it can be the case to find a collection containing objects of type `person` and of type `student` or `teacher`.

- There can be extended the typing constraints (see section 4.3) to refer not only to attributes, but also to methods and to take into account not only the number of properties, but some specified properties.

- There can be developed an algebra for binary collections. There can be done some modifications to the framework in order that all operations on collections apply also to binary collections. From the specific operations for binary collections there were implemented only *domain* and *range*. But, there can be implemented also *domain restriction* and *subtraction*, *range restriction* and *subtraction*, *inverse*, *composition*, *nest*, *unnest*, *division* and *closure*.

- It was implemented *subcollection constraint*, but it can also be implemented *cover constraint*, *disjoint constraint*, *partition constraint* and *intersect constraint*.

- Currently, a collection must be of bulk type *set*. The framework can be extended for supporting also bulk types *bag*, *ranking* and *sequence* behaviour.

- The XD framework can be modified so that insertion order is kept. Suppose we create three objects of type `teacher` and insert them into collection `Teachers`. The following XD DML statements create three objects and insert them into collection `Teachers`:

```
create object teacher1;
insert into collection Teachers:[teacher1];
create object teacher2;
insert into collection Teachers:[teacher2];
create object teacher3;
insert into collection Teachers:[teacher3];
```

Suppose we want to iterate over the members of collection `Teachers`. The current version of XD framework retrieves the elements of collection `Teachers` in a random order. For example, the first element

is object `teacher3`, the second is object `teacher1` and the third is `teacher2`. The XD framework can be modified so that the order of iteration of the members of the collection is the order of insertion, i.e. `teacher1`, `teacher2` and `teacher3`.

- We have designed heterogeneous collections without specifying any constraints on the types the members of the collection can be dressed with. In this way, we have given a total freedom to the user to decide what objects to insert in a collection. We can make the restriction that the objects of a collection have a given set of common properties. In this way we take care to do not group into the same category objects having nothing in common.

6.2 Conclusions

In this report we have presented a framework capable of supporting Rapid Information Modelling. Our prototype application has shown that it is possible to use our framework for developing a Web applications which offer flexible access to information. The management of information is done at the conceptual level without the necessity of a user to define a data model for information organisation.

We have seen that, for allowing rapid information modelling, it was necessary to introduce the notion of heterogeneous collections as well as to provide algebra operations which can be evaluated with or without type checking. Also, it was defined a flexible notion of typing. One can easily define new typing constraints according to user needs and integrate them into the framework.

A developer can use our framework for implementing RIM applications very easily. Further, it is possible to use a variety of database management systems (DBMS) for storing data since our framework is based on the XD framework which is independent of a particular DBMS.

Bibliography

[Atk89] M. Atkinson, F. Bancilhon, D. Dewitt, K. Ditrich and D. Maier. The object-oriented database system manifesto. *Proc. of* 1_{st} *International Conference on Deductive and Object-Oriented Database Management Systems*, Kyoto, Japan, December 1989

[BCN92] Carlo Batini, Stefano Ceri and Shamkant B. Navathe. *Conceptual Database Design. An Entity-Relationship Approach*, Benjamin/Cummings Publishing, 1992

[BMS84] Michael L. Brodie, John Mylopoulos and Joachim W. Schmidt, *On Conceptual Modelling. Perspectives from Artificial Intelligence, Databases and Programming Languages*, Springer-Verlag New York Inc. 1984

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Design*, Prentice-Hall, 1991

[Dol98] Robert Dollinger. *Baze de date si Gestiunea Tranzactiilor*. Microinformatica, Cluj-Napoca, 1998

[DT93] Tharam Dillon and Poh Lee Tan. *Object- Oriented Conceptual Modelling*. Prentice Hall, 1993

[EG97] David W. Embley and Robert C. Goldstein. *Conceptual Modeling-ER'97*, 16^{th} International Conference on Conceptual Modeling, Los Angeles, California, USA, November 1997, Proceedings, Springer-Verlag Berlin Heidelberg, 1997

[EN94] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*, Benjamin/Cummings Publishing, 2 edition, 1994

[Goo99] James Goodwill. *Developing Java Servlets*. Sams Publishing, 1999

[Kob00] Adrian Kobler. *The eXtreme Design Approach*. Draft Thesis, 2000

[KN00] Adrian Kobler, Moira C. Norrie. *OMS Java: An Open, Extensible Architecture for Advanced Application Systems such as GIS*. International Workshop on Emerging Technologies for GEO-Based Applications, Ascona, Switzerland, 2000

[Mod92] Martin E. Modell. *Data Analysis, Data Modelling, and Classification*. McGraw-Hill, 1992

[Mos99] K. Moss. *Java Servlets:second edition*. McGraw-Hill, 1999

[Nor95] M.C. Norrie. Distinguishing Typing and Classification in Object Data Models. In *Information Modelling and Knowledge Bases*, vol VI, ch.25 IOS, 1995 (originally appeared in Proc. European-Japanese Seminar on Information and Knowledge Modelling, Stockholm, Swedan, June 1994)

[NW99] Moira C. Norrie and Alain Würgler. *OMS Pro Introductory Tutorial*. Institute for Information Systems, ETH Zurich, March 1999

[SKN98] Andreas Steiner, Adrian Kobler and Moira C. Norrie. OMS/Java: Model Extensibility of OODBMS for Advanced Application Domains. In *Proc* 10^{th} *Conf. on Advanced Information Systems Engineering (CAISE'98)*, Pisa, Italy, June 1998

[SW94] Douglas Schenck and Peter Wilson. *Information Modeling the EXPRESS Way*. Oxford University Press, 1994

[Wir96] N. Wirth. *Compiler Construction*. Addison-Wesley, 1 edition, 1996

[Servl] http://www.java.sun.com/products/servlet/2.2