# $M^4$ -The **M**ining **M**art **M**eta**M**odel

Anca Vaduva[‡] Jörg-Uwe Kietz[†] Regina Zücker[†] Klaus R. Dittrich[‡]

[‡] University of Zurich
Dept. of Information Technology
CH-8057 Zurich, Switzerland
{vaduva,dittrich}@ifi.unizh.ch

[†] Swiss Life
IT Research & Development
CH-8022 Zurich, Switzerland
{Uwe.Kietz,Regina.Zuecker}@swisslife.ch

**Abstract**

Today's data mining algorithms and tools have specific input requirements which inherently demand preparation of data before their use. As a consequence, one of the most time-consuming steps in the process of knowledge discovery in databases (KDD) is data preprocessing, i.e., preparing data for data mining. Common preprocessing operations include the construction of new features derived from existing ones, adjustment of data formats, data segmentation, sampling and cleansing. The Mining Mart project proposes a case-based reasoning approach that enables both automatization of preprocessing and reusability of defined preprocessing cases for data mining applications. The system architecture follows a *metadata-driven* software approach. This paper mainly deals with the structure of the metadata to be stored. This metadata structure is called *metamodel* and is the core of the system since all components have to be built in accordance with it.

The metamodel considered in this paper ($M^4$) has been developed as a collaboration between two projects, Mining Mart[1] and SMART[2]. The latter project deals with metadata management for data warehousing and considers metadata globally, with a focus on metadata integration. The aim of SMART is an enterprise-wide metadata management system that consistenly and uniformly manages all metadata available in a company in order to provide better support for complex data warehousing processes. In this context, $M^4$ may be seen as part of the global metamodel behind the SMART metadata management system.

## 1 Introduction

Extracting information and knowledge from data is the purpose of advanced technologies like data mining, data warehousing and information retrieval. Data mining combines statistical and mathematical techniques with machine learning algorithms and other artificial intelligence approaches and aims at detecting

[1] http://www-ai.cs.uni-dortmund.de/FORSCHUNG/PROJEKTE/MININGMART/index.eng.html
[2] http://www.ifi.unizh.ch/dbtg/Projects/SMART/

unknown patterns in data. This knowledge is then used for supporting business analysis and trend prediction. Even though a significant amount of algorithms and tools is available on the market, data mining is a complex task. It needs to be embedded in a comprehensive process, called knowledge discovery in databases (KDD) [2]. Data has to be first collected, selected, integrated, cleaned, and then preprocessed in order to fullfil the input requirements of the chosen data mining tool or algorithm. Preprocessing operations include data transformations (e.g., data type conversion), aggregation, scaling, discretization, segmentation, sampling [5]. Practical experiences [9] have shown that 50-80% of the efforts for knowledge discovery are spent for data preprocessing which is not only time-consuming but also requires profound business, data mining and database know-how.

In this context, the aim of the Mining Mart project is to provide a user-friendly environment for performing preprocessing for data mining. To this end, a *case-based reasoning framework* has to be built [5]. The framework provides a collection of *cases* and tools to design these cases. A case consists of the specification of a mining task (e.g., selecting suitable addresses for a mailing action), the data to be mined, i.e. the population, and a chain of preprocessing operators to be applied to this population. Each mining task deploys a certain mining tool or algorithm with special input requirements (see Figure 1) and thus the target of the preprocessing chain is data prepared in accordance with these requirements.

A defined case may be either directly executed or reused for developing new ones: on the one hand, an end-user without any data mining and database knowledge may retrieve one of the prepared cases, make some simple adaption if required (e.g., the selection of a different population) and initiate the case execution. On the other hand, the highly skilled power-user (i.e., the KDD-expert) may use the framework for creating new cases. To this end, he reuses building blocks (i.e., operators) or parts of the chains available from the already defined cases.

One of the particularities of the Mining Mart approach is the implementation of the framework as *metadata-driven* software (see next section). This solution enhances reusability and flexibility of the system.

The remainder of this paper is organized as follows: the next section explains the notion of metadata and metadata-driven software in order to better understand Section 3 which discusses general aspects of the system architecture. In Section 4 we present an overview of the metamodel of the repository, $M^4$. Section 5 and Section 6 describe the metamodel in more detail: each individual class with its attributes, associations and restrictions is presented; Section 5 deals with the data modelling part while Section 6 addresses the case modelling part of $M^4$. Section 7 concludes the paper.

## 2 Metadata-Driven Software

In information systems area, metadata (data about data) is a general notion that captures all kind of information necessary to support the management, query, consistent use and understanding of data. In particular, metadata may be any information related to schema definitions and configuration specifica-

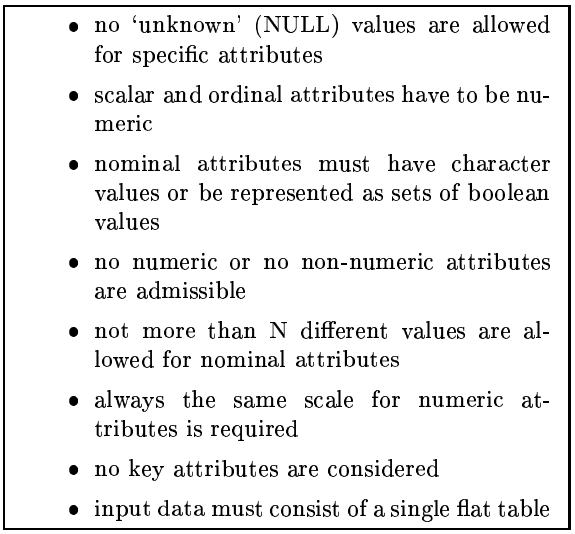


Figure 1: Input restrictions of data mining tools [5]

tions, physical storage, access rights, etc. Metadata may also represent end-user-specific documentation, dictionaries, business concepts and terminology, details about predefined queries, and user reports. Overviews of the state-of-the-art in metadata management with a focus on data warehousing are provided in [10, 11].

In the case of a *metadata-driven software* package, metadata is stored in a repository and is used as *control information* for applications implemented with this software package. Examples of control information are static information (like structure definitions, configuration specifications, etc.) as well as some parts of application logic: conditions (e.g., for dynamic SQL), methods, or parameters for stored procedures. At runtime, metadata is read by a tool engine, is dynamically bound into the engine software and the resulting application is then executed. In other words, application semantics is simply distributed between the repository and the engine and is pieced together at runtime only. Examples of metadata-driven software are the new generation tool packages for data warehousing, e.g., for building the data warehouse (like PowerMart[3], Ardent[4]) or for using it (like Cognos[5], Business Objects[6]).

To summarize, metadata-driven software provides a framework consisting of a repository structure and an engine which fits this structure. Users have to specify metadata instances (i.e., to fill in the repository in accordance with this structure) in order to achieve executable task-oriented applications.

One of the main benefits expected from metadata-driven software is *reusability and flexibility*. On the one hand, objects encapsulating control information are explicitely stored in the repository (instead ob being hidden in scripts and programs) and may be reused in different contexts and applications. On the other hand, the engines running on top of the repository may be used for all metadata instances fitting the given metadata structure. This results in im-

[3]http://www.informatica.com/

[4]Ardent Software was recently acquired by Informix, http://www.ardentsoftware.com

[5]http://www.cognos.com

[6]http://www.businessobjects.com

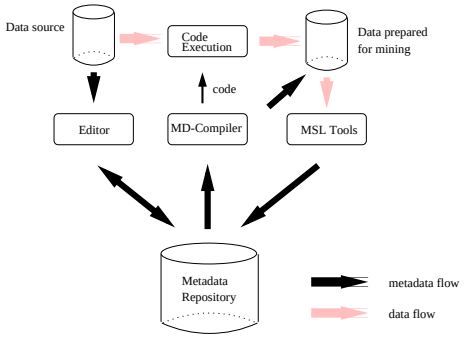


Figure 2: Architecture of Mining Mart System

proved flexibility. The system may be extended and adapted without difficulty. If new requirements arise, metadata instances may be easily changed without affecting the clients (i.e., engines) sharing it. Thus, *maintenance* is easier. Moreover, since operational metadata is for sure kept up-to-date, the documentation of the system is implicitly up-to-date as well.

Nevertheless, the main advantage of using metadata-driven software is when enterprise-wide metadata integration [12] is considered. Metadata stored in various repositories (e.g., from various tools like those for building a data warehouse resp. for using it) is integrated and linked with each other such that it is consistenly and uniformly managed by an enterprise-wide metadata management system. In this way, links between metadata of various domains are established and exploited and thus up-to-date system information and documentation is available to all users and tools across the enterprise. Efforts are underway to establish metamodel standards for enterprise-wide metadata integration and exchange (for a comparison between two important standard proposals see [13]).

## 3 Architecture of the Mining Mart System

Mining Mart follows a typical metadata-driven software architecture, depicted in Figure 2. The core of the system is the *Repository* which is implemented on top of a DBMS. Case-specific information is stored in the repository: the specification of the business problem to be solved by the case, the specification of structures of the data to be mined, the specification of processing operators to be applied on the data with corresponding parameters, the description of the data mining tool for which the data has to be prepared, etc. At runtime, a MD-*Compiler* reads these metadata and uses them in order to generate code. When executing this code, data is read from the data source (e.g., a data warehouse), is preprocessed and stored into the target system on which data mining will be applied later. The *Editor* is used for manipulating metadata (insert, delete, update) within the repository.

Note in Figure 2 that metadata may be produced and stored into the repository by means of other components as well. This component represents the *MSL tools* (multistrategy learning tools) [5] which are used for determining operator parameters when these cannot be manually specified. MSL tools accompany manual preprocessing operators and produce the metadata they require, i.e., the input parameters for them. A typical example is the discretization opera-

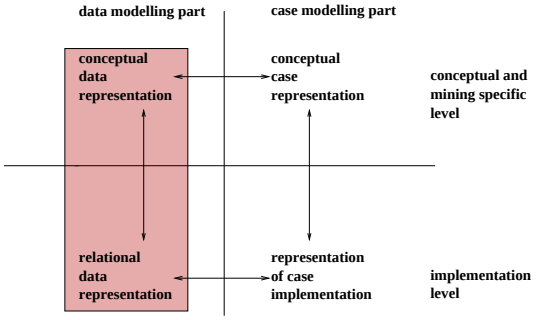


Figure 3: Coarse Description of the Metamodel (the marked rectangle denotes the part of $M^4$ that considers data representation)

tor. One of the input parameters is a discretization table which specify for a given attribute, value intervals and their corresponding discrete values. During preprocessing, the discretization operator checks whether an attribute value is in the range of given intervals and substitutes it with the corresponding discrete value of the input table. Since the manual specification of an optimal discretization table is not always feasible, an MSL tool has to be used for discovering the best discretization table by means of data analysis. When optimal parameter settings depend on data, their discovering by means of MSL tools is the prerequisite for case reuse. For example, if discretization tables are automatically rendered by a tool, the same case may be directly re-executed without designer intervention for different populations.

The software components accessing the repository (Editor, MD-Compiler, MSL-Tools) are "bound" to the given metadata structure which is conceptually described by a *metamodel*. The domain-specific language for specifying applications is derived from the metamodel as well. The next section introduces the Mining Mart metamodel which is then described in more detail in the remainder of the paper.

# 4 Overview of $M^4$

We now give a coarse description of the metamodel of Mining Mart with a focus on the data representation part. Since a metamodel should "catch" the particularities that are relevant for a specific domain, our data representation part reflects the features of the data set that are important for preprocessing.

The Mining Mart Metamodel ($M^4$) is illustrated as a class diagram in the Appendix. $M^4$ can be logically divided into two main parts, one managing information with regard to *data modelling* and the other one regarding *case modelling*. Each part is again subdivided in accordance with the abstraction level into *conceptual and mining specific* representation on the one hand and *implementation* representation on the other hand. Figure 3 depicts the four parts resulting from this partition; they are tightly coupled to each other.

- *Data modelling* part comprises classes for describing the *relational data representation*, which corresponds to the implementation level and the *conceptual data representation* which essentially deals with the entity-relationship model enhanced with data mining specific aspects (as e.g.,

special data types - Time, Ordinal, Nominal, etc) and ontology knowledge of the application domain.

- *Case modelling* part describes preprocessing operators and the required controlling structures. This submodel is again divided into the mining-specific description of the case semantics (including for example operators like feature selection and discretization) and their implementation as e.g., function, stored procedure or SQL-query. We call the two metamodel parts *conceptual case representation* and *representation of the case implementation* respectively.

On the one hand, partitioning $M^4$ in data vs. case modelling representation is necessary for ensuring reusability: the already specified operators may be used within cases which have parameter values represented in the data modelling part. Cases may be reused for different data sets (i.e., populations), represented within the data modeling part. On the other hand, distinguishing between the two abstraction levels (conceptual vs. implementation) is required for enhancing

- *user-friendliness*: End-users may manipulate familiar elements on the conceptual level in order to configure cases for execution. Regarding technical users, the two main categories are *case designer* and *case adapter*. The case designer accesses and manipulates only the elements of the upper, conceptual level during his work. Thus, the implementation is transparent to him: he deals with mining-specific elements and constructs and has not to be aware of how they are implemented (which DBMS is used, how functions are implemented, etc.). In contrast, the case adapter is responsible for building the connection between the two levels when the structure of the database changes.

- (again) *reusability*: The conceptual, abstract level may be (re)used independently on the actual implementation of the database or of the operators. A relational data model has been chosen for data storage on the implementation level. That means, the input and output data will be stored in a relational database system (we considered Oracle). However, for the same specification on the conceptual level, also other data models may be used on the implementation level (e.g., hierarchical, object-oriented model, etc.).

- *transportability* (which is a sort of reusability as well): The idea is to be able to reuse (parts of) the cases not only in the same company (e.g., Swiss Life) and the same branch (life insurance), but in other branches as well. To this end, the representation of ontologies [3, 4, 8] has to be considered within the *conceptual data representation* part. A common ontology basis has to exist which is then specialized by domain-specific ontologies.

The four parts of $M^4$ are linked to each other and connections between metadata instances are often navigatable in both directions such that the required information may be rapidly accessed. Note that the consideration of the case implementation submodel (the quarter right below) is optional. Since this part represents detailed information related to the implementation of operators, it

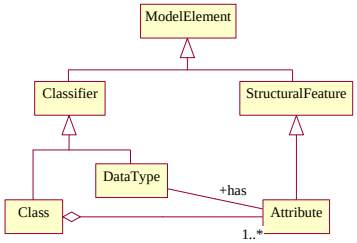


Figure 4: Simplified UML

makes sense only if a step-by-step tracing of data transformations is intended. In other words, the case implementation submodel is relevant only if pieces of code corresponding to the execution of operator are broken in smaller pieces which have to be extensively documented. These pieces need to be linked with the data modelling part where the input and output data elements are explicitely stored. This could be desired for supporting understanding, debugging and maintenance of code. Otherwise, if implementation information is stored with a coarse granularity only (e.g., to an operator corresponds an atomic function call), the two case levels, conceptual case and implementation case representation are merged.

$M^4$ combines ideas from two existing standards for metadata representation and exchange in the area of data warehousing (OIM and CWM) [13]. They are drastically simplified but extended with data mining and preprocessing elements to make the metamodel domain-specific. Since both standards have the metamodel of UML as their core, $M^4$ uses some UML classes as foundation as well. That means, UML is not only used as (graphical) language for describing class diagrams but it is also used as the core metamodel, extended within $M^4$. That means, UML classes are specialized in $M^4$. For describing the foundation classes and the classes of $M^4$, we adopt the following description format: class name, brief description, supertypes, class attributes and associations with other classes (including association multiplicity).

**Foundation of the Metamodel: UML Classes**
Figure 4 depicts the UML classes which are specialized for defining $M^4$. These are:

**ModelElement** is the base for all modeling metaclasses in the UML. All other modeling metaclasses are either directly or indirectly specialized from ModelElement.

**Classifier** A classifier is a general element that describes behavioral and structural features; it appears in several specific forms, that means, the class Classifier may be specialized as Class, DataType, Interface, Component, aso.

**Class** A Class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. Specializations in $M^4$ of the UML class *Class* are ColumnSet and Concept (supporting conceptual and relational modelling).

**Supertype** Classifier
**Subtypes** (in $M^4$) Concept, ColumnSet
**Associations** *attributes*: points to the corresponding set of class attributes. Multiplicity: 0..n.

**Attribute** An attribute is a named slot within a classifier that describes a range of values that can be assigned to instances of the classifier. Attribute is specialized to support different needs, and associated with a data type.
**Supertype** StructuralFeature
**Subtypes**(in $M^4$) Column, FeatureAttribute, Value, RoleRestriction
**Associations** *classes*: is an association to the class this attribute is in. Multiplicity:1..1.
*dataType*: points to the DataType of this attribute. Multiplicity: 1..1.

**DataType** A data type is attached to an attribute. Data types include primitive built-in types (integer, strings etc.) as well as definable enumeration types (e.g. boolean, true and false).
**Supertype** Classifier
**Subtypes**(in $M^4$) Integer, String, DomainDataType etc.
**Associations** *has*: points to the Attributes having this data type. Multiplicity: 0..n.

In the following, we present $M^4$ in more detail. The next two sections deal with the two submodels obtained through vertical partition (data resp. case modelling part). We present each class in turn with some of its particularities and we do not make explicit distinction between class and its instances - it is visible from the context whether class or instance is meant. Even if not always explicitly stated, each instance of the classes in the metamodel has a unique *ID* which identifies it, a *name* and a *description* (which is simple text). Moreover, each class manages its *extension* which contains the set of all instances of this class.

# 5 The Mining Mart Data Representation

The data modelling part of $M^4$ is illustrated in Figure 5. It consists of the conceptual and the logical data representation which are strongly coupled with each other. The logical level follows the relational data model which is known from other metamodels as well (e.g., [13]). We start with its description.

## 5.1 Relational Data Representation

This submodel comprises classes for representing data structures that use the relational data model. The main classes are: *Column*, *ColumnSet* (with its subclasses *Table*, *View*, *Snapshot*) and *Key*. *Column* and *ColumnSet* have each a class containing statistical information (*ColumnStatistics* and *ColumnSetStatistics*). A *ColumnSet* consists of a list of *Column*s.

### 5.1.1 Column

An instance of the class Column defines a set of values in a result set, e.g., a view or a table. All values of the same column are of the same data type. A value

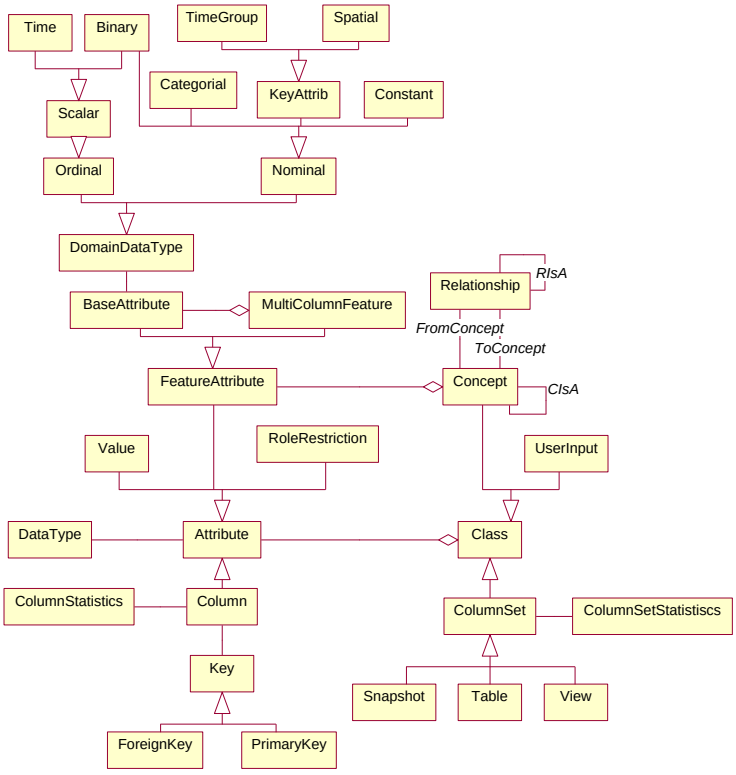


Figure 5: The class diagram of Mining Mart data representation

from a Column is the smallest unit of data that can be selected from a table or view and the smallest unit of data to be updated. A Column corresponds to a BaseAttribute at the conceptual level.

**Supertype** Attribute

**Subtypes** None

**Attributes**

- *name*: name of the column in the database schema (e.g., PTANSCH, PTEPFI, etc.).
- *dataType*: data type in the implementation language of the column (e.g., integer, string, etc).

**Associations**

- *belongsToColumnSet*: an aggregation[7] to ColumnSet (i.e., Column is part of ColumnSet). It points to the ColumnSet that contains this Column. Multiplicity: 1..1.
- *keys*: an association describing in which keys this column is part of. Multiplicity: 0..n.

[7]Aggregation will be implemented as an attribute TableIndentifier being foreign key to ColumnSet.

- *correspondsToBaseAttribute*: an association pointing to the corresponding BaseAttribute at the conceptual level. Note that a FeatureAttribute may be either a BaseAttribute or a MultiColumnFeature. In the latter case there are more than one corresponding columns.

### 5.1.2 ColumnSet

A ColumnSet describes any general set of columns - typically a table, view or snapshot.

**Supertype** Class

**Attributes**

- *name*: name of the table, view, etc.
- *number*: number of columns.
- *file*: name of the file containing the command creating the ColumnSet.
- *dbConnectString*: name of the database where the ColumnSet belongs to.
- *user*: the name of the owner of the ColumnSet (e.g., for the access in Oracle User.Name@DBString is needed).

**Associations**

- *hasColumn*: an aggregation over Columns (ColumnSet has Column). It points to all the Column(s) that form this ColumnSet. Multiplicity: 1..n.
- *hasKeys*: an association to the corresponding primary and foreign keys that apply to the ColumnSet. Multiplicity 1..n.
- *correspondsToConcept*: an association to the corresponding concept at the conceptual level. Multiplicity 0..1.
- *correspondsToRelationship*: an association to the corresponding relationship at the conceptual level. Multiplicity 0..1.

**Constraints** ColumnSet points either to a Concept or a Relationship. There are ColumnSets which do not point to any concept but to a relationship - this is the case when the Relationship has the multiplicity m:n, then it will be implemented as a separate table.

### 5.1.3 ColumnStatistics

ColumnStatistics contains statistic information for columns necessary during data mining. This information includes statistics of the values for each Column (e.g., maximal and minimal value, average, etc) but also the distribution blocks necessary for further preprocessing (if such information is available). Distribution blocks contain values grouped in accordance with some criteria. Mining specific types play an important role when building distribution blocks: for *nominal* attributes, every value is counted and those with the same weight are grouped. For *ordinal* attributes, values are grouped according to certain intervals (at most 1000 intervals). For *time* attributes the distribution usually

depends on the number of months between the minimal and maximal value according to the following rules:

| | |
|---|---|
| month_number > 600 | ⇒ values are grouped into years |
| 60 < month_number <= 600 | ⇒ values are grouped into quarters |
| 3 < month_number <= 60 | ⇒ values are grouped into months |
| months_number <= 3 | ⇒ values are grouped into days |

**Supertype** ModelElement

**Attributes**

- *unique*: number of different values of this column within the ColumnSet.
- *missing*: number of missing value(s) within the ColumnSet.
- *min*: minimal value of the column within the ColumnSet.
- *max*: maximal value of the column within the ColumnSet.
- *average*: average value of the column within the ColumnSet.
- *standardDeviation*: standard deviation value of the column within the whole ColumnSet.

The last two attributes make sense for numeric (i.e., scalar) attributes only. The distribution information consist of:

- *distributionValue*: name of one distribution block, e.g 'YOUNG' for the attribute 'AGE'. If the attribute is of the type ordinal, the average value of the block is used.
- *distributionCount*: number of counted records for this distribution block.
- *distributionMin*: minimal value of the distribution block (makes sense for ordinal attributes only).
- *distributionMax*: maximal value of the distribution block (makes sense for ordinal attributes only).

- **Associations**
  - *forColumn*: an association that points to exactly one Column. Multiplicity: 1..1.

**Note** Statistics could be represented at the conceptual level as well.

### 5.1.4 ColumnSetStatistics

ColumnSetStatistics contains statistic information for ColumnSets. For each instance of ColumnSet, the number of columns having the same mining specific data type (e.g., ordinal, nominal time) has to be stored.

**Supertype** ModelElement

**Attributes**

- *allNumber*: total number of tuples within the ColumnSet.
- *ordinalNumber*: number of ordinal attributes of the ColumnSet.

- *nominalNumber*: number of nominal attributes of the ColumnSet.
- *timeNumber*: number of time attributes of the ColumnSet.

**Associations**

- *forColumnSet*: an association pointing to exactly a ColumnSet. Multiplicity: 1..1.

### 5.1.5 Table, View, Snapshot

These classes represent the notions of table, view and snapshot known from implementation data models of database management systems (like Oracle). Their superclass is ColumnSet. View and Snapshot have an attribute containing the filtering condition (the WHERE part of the SQL- query used for view definition) and one association pointing to one or more instances of ColumnSet it has been applied on (FROM part of the view definition). Snapshots require also attributes for updating as *howRefresh* (i.e., either FAST or COMPLETE) and *refreshInterval*.

### 5.1.6 Key, PrimaryKey, ForeignKey

The *PrimaryKey* and *ForeignKey* classes represent the corresponding notions known from the relational model. The class *Key* is the superclass of *PrimaryKey* and *ForeignKey* and is an abstract class.

**Supertype** (applies for *Key*) ModelElement

**Attributes**

- *isUsedForIndex* (applies for *PrimaryKey*): a boolean attribute which may take two values, yes or no.

**Associations**

- *hasColumn*: an association to the Column(s) that form the Key. Multiplicity: 1..n.
- *isAssociatedToColumnSet*: an association to the ColumnSet where it is Key (it is a sort of redundancy to the aggregation between Column and ColumnSet).
- *isConnectionTo* (it exists for ForeignKeys only): an association to the Table where is key (usually a primary key). Multiplicity 0..1.
- *correspondsToRelationship* (it applies for ForeignKey): an association pointing to the corresponding relationship at the conceptual level. Multiplicity 0..1. Relationships 1:m or 1:1 will be implemented as ForeignKeys. A relationship m:n will be implemented as ColumnSet and 2 ForeignKeys. Multiplicity 1,2 or more.

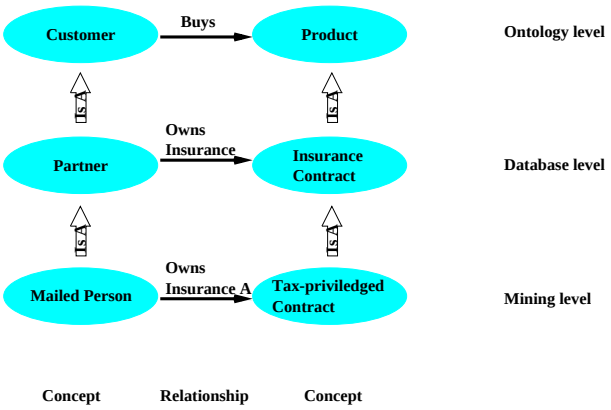


Figure 6: Instances of Concept and Relationship

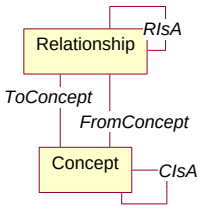


Figure 7: The classes *Concept* and *Relationship* and their UML associations

## 5.2 Conceptual Data Representation

Due to the significant role it plays for user-friendliness and reusability, the conceptual layer is the most important part for the data representation in $M^4$. It is essentially Entity-Relationship (ER) data representation enhanced with description logic (DL) and ontology representation, extended with data mining specific features. The main two classes are *Concept* and *Relationship*. A Concept (e.g., *Customer*, *Partner*) may have subconcepts (e.g., *Partner between 30-40 years* or *Mailed Person*), that means there are *IsA* relationships between Concepts (see Figure 6). Application-specific Relationships exist as well (e.g., *Buys*, *Owns Insurance*, *Owns Insurance A*; they are binary Relationships (as in DL), e.g., Concept *Customer Buys* Concept *Product*, Concept *Partner* is in Relationship *Owns Insurance* with Concept *Insurance Contract*. Relationships may be bound to each other with *IsA* relationships as well. Figure 7 illustrates these semantics as represented in the metamodel.

As shown in Figure 6, the three perspectives covered in the conceptual data representation are:

- the *ontology level* contains general business ontologies; is useful for reuse of cases in different companies; *Mailed Person IsA Partner* which in turn *IsA Customer*, *Tax-priviledged Contract IsA Insurance Contract* and *Insurance Contract IsA Product*; *IsA* applies for Relationships as well. *Customer* and *Product* are included in the basis ontology which should be common to many companies.

- the *database schema level* represents the conceptual schema of the database;

this is mapped to the implemented schema. In Figure 6 this level corresponds to *Partner, Insurance Contract* and *Owns Insurance* which have direct correspondents on the implementation level in terms of relational tables. So far, only the relational model is considered on the implementation level in $M^4$ (see Section 5.1) but other data models may be considered as well. Note that the basis ontology (*Customer*, *Product*, *Buys*) has no direct correspondence on the implementation level.

- the *mining data level* describes data sets needed for and produced during preprocessing for mining; contains also mining specific data types. In Figure 6 these are the *Mailed Person*s, *Tax-priviledged contract*s, and the Relationship *Owns Insurance A*. These are subconcepts and subrelationships of the elements beyond and are directly used for configuring or designing Mining Mart cases. They correspond on the implementation level to views or snapshots on tables.

We consider in turn the classes of the conceptual data representation with their attributes.

### 5.2.1 Concept

A Concept is the basic element of the conceptual data representation. It expresses a "thing" in the application domain.

**Supertype** Class, Parameter

**Attributes**

- *name*: name of the concept (e.g., Partner, Product, Customer).
- *subConceptRestriction*: specification (using a machine processable language!) of the characteristics of a subconcept (in relation with its superconcept), e.g., the specification of the fact that concept *Young&Powerful* represents the *Partner*(s) between 30-40 years while *Young* represents *Partner*(s) between 20-30 years. Both are subconcepts of *Partner*.

**Associations**

- *isA*: an association to the (super)concept e.g, *Young* isA *Partner*.
- *correspondsToColumnSet*: an association pointing to the corresponding ColumnSet that implements the concept. Multiplicity 0..1.
- *FromConcept*: an association pointing to the Relationship the Concept is linked with. For example, Concept *Partner* is in Relationship *HasPartnerRole* to the Concept *Contract*. Then *Partner* is associated by means of *FromConcept* to *HasPartnerRole*. Multiplicity 1..1.
- *ToConcept*: an association to the Relationship the Concept is linked with. E.g.,*Contract* is associated by means of *ToConcept* to Relationship *HasPartnerRole*. Multiplicity 1..1.

**Constraints** Each ColumnSet has a Concept or Relationship but not each Concept or Relationship points to a ColumnSet (e.g., the basis ontology has no correspondant on the database side.
For each instance of *FromConcept* association an instance of *ToConcept* association has to exist and conversely.

### 5.2.2 Relationship

A Relationship expresses the connection existing between two concepts.

**Supertype** ModelElement, Parameter

**Attributes**

- *name*: name of the relationship (e.g., HasPartnerRole, IsInsuredPerson, IsInsuranceHolder, OwnsInsurance).
- *subRelationshipRestriction*: specification (in a machine processable language!) of the characteristics of a subrelationship in relation with its superrelationship, e.g., the specification of the fact that Relationship *IsInsuredPerson* resp. *IsInsuranceHolder* represents a subset of *HasPartnerRole* fulfilling some conditions. These are formulated using FeatureAttributes, Concepts, and so on. A possible language could be DL role-terms [1]. *IsInsuredPerson* and *IsInsuranceHolder* are both subrelationships of *HasPartnerRole*.
- *defined*: an attribute that specifies whether the Relationship is defined according to the restriction above or primitive (sufficient condition or not).

**Associations**

- *isA*: an association pointing to its (super)relationship e.g, *IsInsuranceHolder* isA *HasPartnerRole*.
- *correspondsToForeignKey*: an association to the corresponding ForeignKey(s) that implement(s) the Relationship. Multiplicity 1..n. Relationships 1:m or 1:1 will be implemented as ForeignKeys. A relationship m:n will be implemented as ColumnSet and 2 ForeignKeys (only binary Relationships like in Description Logic are considered).
- *correspondsToColumnSet*: an association to the corresponding ColumnSet that implements the Relationhip. This applies only when the relationship is m:n. Multiplicity 0..1.
- *FromConcept*: an association to one of the Concepts connected by the Relationship. Multiplicity 1..1.
- *ToConcept*: an association to the other Concept. Multiplicity 1..1. E.g., *HasPartnerRole* is associated by means of *FromConcept* to Concept *Partner* and by means of *ToConcept* to Concept *Contract*.

**Constraints** Each Relationship corresponds to either a ColumnSet or a ForeignKey on the implementation level. That means, the association *correspondsToForeignKey* may exist without the association *correspondsToColumnSet*. However, when *correspondsToColumnSet* exists, it requires

two ForeignKeys as well (i.e.,*correspondsToForeignKey* exists as well). A Relationship cannot exist without FromConcept and ToConcept.

### 5.2.3 FeatureAttribute

Contains features of Concepts. It may be either a *BaseAttribute* or a *MultiColumnFeature*.

**Supertype** Attribute (from UML), Parameter

**Subtypes** BaseAttribute, MultiColumnFeature

**Attributes**

- *name*: name of the feature attribute. The name should be more comprehensive (e.g., postal address of partner) than usual names of columns (as. e.g, PTANSCH and PTEPFI).
- *relevanceForMining*: a boolean attribute expressing whether the FeatureAttribute is relevant for mining or not.
- *attributeType*: an attribute which depicts whether it is a base attribute, a result or an intermediate attribute produced during case execution.

**Associations**

- *belongsToConcept*: an aggregation to the concept it belongs to. Multiplicity 1..1.
- *correspondsToColumns*. an association to the column (or columns) it represents. Multiplicity 1..n. Note that a feature attribute may have more than one column if it is a MultiFeatureColumn.

### 5.2.4 BaseAttribute

A *BaseAttribute* may have various mining relevant data types (they follow below).

**Supertype** FeatureAttribute

**Associations**

- *domainDataType*: an association to the mining-specific data type, i.e., to the DomainDataType. Multiplicity 1..1.
- *isPartOfMultiColumnFeature*: an aggregation to a MultiColumnFeature if it is part thereof. Multiplicity 0..1.

### 5.2.5 MultiColumnFeature

A *MultiColumnFeature* consists of a set of *BaseAttribute*s. Note that a MultiColumnFeature has no data type, only a BaseAttribute has one.

**Supertype** FeatureAttribute

**Subtypes** *TimeInterval* (not depicted in the diagram)

**Associations**

- *consistsOfBaseAttributes*: an aggregation representing the set of BaseAttributes that builds the MultiColumnFeature. Multiplicity 1..n.

**Note** *TimeInterval* has only two BaseAttributes it points to: *startOfInterval* and *endOfInterval*. Their data type is *Time*. Another example of a MultiColumnFeature is "Money" which could be represented with two components (value, currency). Note that, in contrast to Time, Money is represented as an instance of MultiColumnFeature, not as subclass.

### 5.2.6 Value

*Value* is needed within arithmetic expressions and conditions used in operators like for e.g., segmentation operators. That means, operators may have parameters that are constants (like e.g., *scalingfactor*) and these constants have to be expressed as Values. Value is part of *UserInput*. This aggregation is not additionally depicted in the figure since superclasses of *Value* and *UserInput* (i.e., *Attribute* and *Class*) are anyway linked by an aggregation.

**Supertype** Attribute (from UML), Parameter

**Attributes**

- *name*: name (or representation) of the value.

**Associations**

- *domainDataType*: an association to the mining-specific data type, i.e., to the DomainDataType. Multiplicity 1..1.
- *belongsToUserInput*: an aggregation to the UserInput it belongs to. Multiplicity 1..n., not depicted in the diagram.

**Note** Values could be any complex structure as well, e.g., decision trees, regression trees, discretization tables, instance lists, aso. Complex structures have not been considered so far.

### 5.2.7 UserInput

Contains the set of *Value*s entered by users when specifying cases.

**Supertype** Class (from UML)

**Attributes**

- *name*: name of user input.

**Associations**

- *containsValues*: an aggregation pointing to Values. Multiplicity 1..n. Not depicted in the diagram.

### 5.2.8 RoleRestriction

It is a special Attribute. It is necessarily bound to a Relationship and a Concept. It actually expresses a constraint, the fact that if a Relationship is linked to a Concept by means of FromConcept, it has to be linked to another Concept by means of a ToConcept. RoleRestriction considers the constraint from another perspective, for THIS (i.e., the given) Concept for which the RoleRestriction attribute has been defined, there exists a Relationship and (at least) another Concept (such that the relationship exists between these two concepts).

Warning: the semantics may possibly be found in the Concept-Relationship modelling as well but it seems that various operators need the information explicitly available in form of "role restriction" and not hidden as Concept-Relationship representation. That is the reason RoleRestriction had to be introduced. It corresponds to the DL terms: all, atleast and atmost [1].

**Supertype** Attribute (from UML)

**Attributes**

- *name*: name of the role restriction.
- *restrictionForRelationship*: a pointer to the Relationship it is a restriction for.
- *restrictionForConcept*: a pointer to the (sub)Concept it applies to.
- *restrictionToConcept*: a pointer to the Concept where all instances of the range of the relation will be member of (DL-all).
- *min*: minimum number of Concept instances[8] in Relationship with every instance of THIS Concept.
- *max*: maximum number of Concept instances in Relationship with every instance of THIS Concept. (Note that Relationships may exist only between two Concepts).

**Associations**

- *belongsToConcept*: an association to THIS Concept for which roleRestriction is an attribute (in the diagram is depicted as an association between Attribute and Class).

### 5.2.9 DomainDataType and its Subclasses

We only briefly mention below the domain-specific data types relevant for data mining and data preprocessing.

**DomainDataType** This class represents the domain specific data type of BaseAttribute. Each BaseAttribute has a data type; this will be represented as an association *hasDataType*, multiplicity 1..1. Generally, note

[8] In this case, "instance" is meant as data object (e.g., "Meier", "contract no. 119725", "contract no. 129726". It is important to make the distinction between data (i.e., "Meier"), metadata (concept "Partner") and metamodel (class "Concept"). "Meier" is an instance of the concept "Partner", which is an instance of the class "Concept".

that constraints have to be implemented for these data types. In particular, operators making sense for each of the domain-specific data types have to be defined and processing information for them is required. For example, "<" and "<=" make sense for ordinal attributes only. In contrast, "=" makes sense for binary and categorial attributes. Distance is allowed for scalar attributes only. Operators like +, - are applied to scalar attributes. Logical operators are applicable to binary attributes.

**Ordinal** All values of this attribute are ordered. Distance between values makes no sense.

**Scalar** Distance makes sense. Scalar attributes are usually represented as numeric or date on the implementation level.

**Time** It represents the absolute point in time. It may have two attributes, *value* and *timeScale* (second, minute, hour, day, month, year).

**Binary** Has only two values, 0 or 1. "<" and distance makes sense (it is either 0 or 1). The two logical operations, "xor"(+) and "and" ($\cdot$) are also applicable and result in the following arithmetic logic:

| XOR: | AND: |
|---|---|
| $1 + 1 = 0$ | $0 \cdot 0 = 0$ |
| $1 + 0 = 1$ | $0 \cdot 1 = 0$ |
| $0 + 1 = 1$ | $1 \cdot 0 = 0$ |
| $0 + 0 = 0$ | $1 \cdot 1 = 1$ |

**Categorial** Has a fixed small number of values.

**KeyAttribute** This kind of attributes is used for identification and is not suitable for mining (the number of different values is too high). Subclasses are *TimeGroup* and *Spatial*.

**TimeGroup** It represents the identification of an individual for which *Time* data is collected. It makes sense only if it is paired with a set of *Time* attributes (representing the time series). It has as attribute *numberOfDifferentIndividualInstantiations* and *missing values*.

**Spatial** This data type is used for geographical information systems (GIS) and mining visualisation.

**Constant** It has only one possible value and thus it is not suitable for mining. Typically is the result of a selection.

# 6 Mining Mart Case Representation

As depicted in Figure 8, classes in this submodel manage metadata for configuring cases, preprocessing operators and metadata for connecting these operators. A case consists of a list of steps and each step embeds an operator; the output of a step is the input for the next one. Operators have *Parameters* which are instances of *Concept*s, *Relationship*s, *FeatureAttribute*s or *Values* - to be found in the data modelling part (not all these semantics are represented in the diagram). Cases have as parameters, among others, the population (i.e., the base

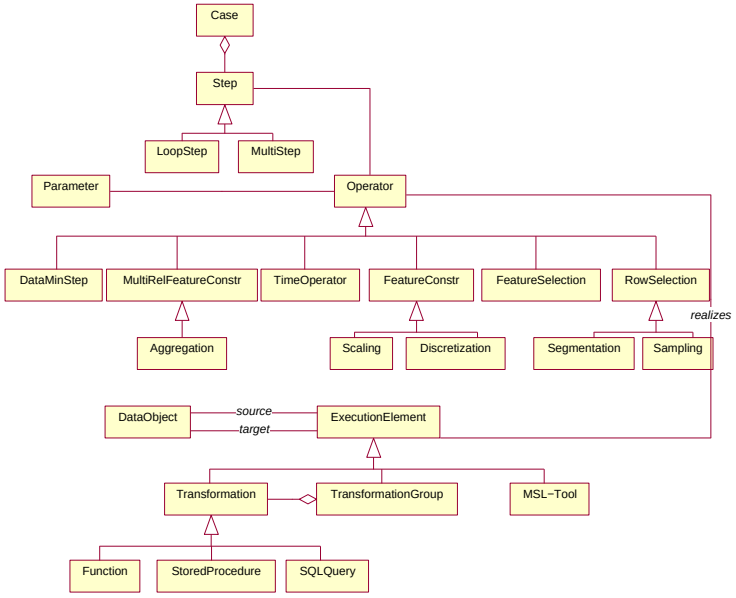


Figure 8: The class diagram of Mining Mart case representation

concept) and the target attribute which need to be specified at the conceptual level. The result of the preprocessing chain is usually a (sub)concept (of the base concept) and one or some FeatureAttributes on which the actual data mining has to be finally performed.

## 6.1 Conceptual Case Modelling

This layer contains domain-specific classes which represent preprocessing on a higher abstraction level than the implementation code behind (represented within the case implementation part in Section 6.2).

### 6.1.1 Case

A case consists of many steps. Note that a case has to contain an attribute *documentation* which describes by means of natural language what the case does. This documentation is important for retrieving the appropriate case from the set of already defined cases.

**Supertype** ModelElement

**Attributes**

- *name*: name of the case.
- *case mode*: a flag denoting whether the case is in training or final mode.
- *caseInput*: a heterogeneous list containing Concepts, FeatureAttributes, Values, Relationships representing the input for the case.
- *caseOutput*: a Concept representing the output of the case; it is actually the input required by the DataMiningStep.

- *documentation*: an attribute containing the description of the case.

**Associations**

- *listOfSteps*: an aggregation of the steps that build the case. Multiplicity 1..n.
- *population*: an association to a *Concept* (which is an element of *caseInput*). The case has to be applied on this Concept; it will be taken as parameter for RowSelection operators[9].
- *targetAttributes*: an association to the *FeatureAttribute*(s) for which the mining case is applied; It will be needed as parameter value for many of the operators.

### 6.1.2 Step

Steps are parts of cases (that means, they make sense only in connection with a case). Each step has to be linked to a set of predecessors and successors which are steps as well. In this way, parallelisation of operator execution is possible because no fixe sequence is enforced, only prioritisation is specified - that means, the operator may be executed only if its predecessors have been executed. Each step embeds one operator. The output of the operator belonging to the predecessor step is needed as the input of the operator belonging to THIS step.

**Supertype** ModelElement(from UML)

**Subtypes** LoopStep, MultiStep

**Attributes**

- *name*: name of the step.
- *description*: an attribute describing what the step does.

**Associations**

- *belongsToCase*: an association pointing to the corresponding Case. Multiplicity 1..1.
- *embedsOperator*: an association pointing to the corresponding operator. Multiplicity 1..1.
- *predecessors*: an association pointing to the preceding steps. Multiplicity 0..n.
- *successors*: an association pointing to the succeding steps. Multiplicity 0..n.

[9]Typically, RowSelection operators have to be used within any case specification.

### 6.1.3 LoopStep

*LoopStep* is a special kind of Step. It allows the iteration along more than one input element (e.g., MissingValue-Operator has to be applied to more than one attributes and this has to be done during the same step). *LoopStep* may be applied only to*Operator*s having *loopable* = yes. These may be only instances of *FeatureConstruction* and *MultiRelationalFeatureConstruction*. Other operators like *RowSelection*, *FeatureSelection*, *TimeOperators* are not loopable. The output description is the same as for a single operator call.

**Supertype** ModelElement (from UML)

**Attributes**

- *iterationSet*: a set of FeatureAttributes; for each element of the set the operator (embedded in step) is called.
- *outputSet*: a set of FeatureAttributes.

### 6.1.4 MultiStep

*MultiStep* is another special kind of *Step*. While *LoopStep* is an iteration over input elements, *MultiStep* is an iteration over output elements. That means, the remainder of the chain preprocessing is applied to each of the output element in the list. In other words, the remainder of DAG (directed acyclic graph) that describes the case for a single output element of this (multi)step has to be repeated for each output element. It results into a multiplication of the DAG's according to the number of output elements.

**Supertype** ModelElement (from UML)

**Attributes**

- *iterationCondition*: the condition for looping over output elements.

**Note** OutputSet is taken over from the Operator embedded in Step.

### 6.1.5 Operator

It is the superclass of all operators in MiningMart.

**Abstract** Yes

**Supertype** ModelElement (from UML)

**Attributes**

- *loopable*: a boolean attribute which may take two values, yes/no. FeatureConstruction and MultiFeatureConstruction could be loopable while the other ones (RowSelection, FeatureSelection, TimeOperators) are not loopable.
- *numberOfInputParameters*: an attribute denoting the number of input parameters for this operator.

- *numberOfOutputParameters*: an attribute denoting the number of output parameters for this operator.
- *manual*: a boolean attribute which may take two values, yes/no. If manual = no then it is a MSL-supported operator and the tool used has to be specified in the next attribute.
- *tool*: an attribute containing the call of the MSL-tool in case the operator is not manual.

**Associations**

- *input*: an association pointing to an ordered list of Parameter. Multiplicity 1..n.
- *output*: an association pointing to an ordered list of Parameter. Multiplicity 1..n.
- *realizes*: an association pointint to an ExecutionElement. Multiplicity 1..1.

Since the special data mining preprocessing operators are not the focus of this paper, we do not consider the operator classes in detail. We only give one example in 6.1.7.

### 6.1.6 Parameter

Parameters are input or output values of Operators and thus may be instances of *Value*, *Concept*, *Relationship* or *FeatureAttribute*s. In this version, we modelled Parameter as the superclass of these classes. Parameter instances may be also updated at runtime (in case they are generated by MSL-tools).

**Supertype** ModelElement

**Subtypes** Value, Concept, Relationship, FeatureAttribute

**Attributes**

- *name*: name of the Parameter.
- *place*: the place of the parameter in the signature of the operator.
- *ParameterType*: the type of the Parameter, it may be Value, Concept, Relationship or FeatureAttribute. Note that Value and FeatureAttribute may have data types themselves.

**Associations**

- *belongsToOperator*: an association pointing to the Operator to which the parameter belongs. Multiplicity 1..1.

### 6.1.7 FeatureConstruction

The operator *FeatureConstruction* creates a new feature for a concept; on the implementation level that means a new attribute in a table or view. The new attribute is based on one or more base attributes. The total number of data records is the same as before the operation. Examples of feature construction

operators are: age computation for a person by using his date of birth, computation of the entry-age into an insurance contract, resp. the end-age of an insurance contract. **Scaling** is a specialization of FeatureConstruction operator. The scale of numeric attributes is significant for distance-based mining algorithms (like e.g., clustering) because attributes with larger values are more influential on the result. To avoid this usually unintended weighting of attributes, all attributes have to be rescaled, i.e., the range of an attribute values has to be changed in such a way that it fits into a specified new range. Input and output parameters have to be scalar *BaseAttribute*s.

**Supertype** Operator

**Attributes**

- *scaling factor*: a number which is either fixed or has to be computed. To compute the scaling factor the domain intervals for input and output BaseAttributes are needed. They have to be read from the data representation submodel.

**Associations**

- *input (inherited)*: an association pointing to a BaseAttribute of type Scalar.
- *output (inherited)*: an association pointing to a BaseAttribute of type Scalar.
- *realizes (inherited)*: an association pointing to the corresponding ExecutionElement which should be a StoredProcedure. This StoredProcedure does the scaling in accordance with the given scaling factor.

## 6.2 Modelling of Case Implementation

The representation of case implementation contains informations needed for algorithms implementing the preprocessing operators. It also contains the links to the data elements which are the input and output for the ExecutionElements implementing the operators. Recall that each operator instance on the conceptual level corresponds to an instance of an *ExecutionElement*. This submodel is necessary only if a detailed tracing of data transformations is intended. In particular, this submodel makes sense only if the aggregation between the *TransformationGroup* and *Transformation* is used (see Figure 9); in this case, the TransformationGroup corresponding to an operator is broken in smaller pieces of code and each piece represents a Transformation. Each Transformation has as input and output a *DataObject* which is either a ColumnSet, Column or a Value. Note that the consideration of case implementation representation causes a proliferation of ColumnSets and Columns because information for (temporary) Columns or ColumnSets produced by any small Transformation has to be stored in the metadata repository. This information may be useful but the developer has to be aware of what it means to collect and store it. In contrast, if each operator is realized by a single, atomic ExecutionElement (be it a StoredProcedure, Function or SQL-Query), the conceptual and

implementation case levels may be merged; the explicit representation of case implementation does not bring advantages anymore.

### 6.2.1 ExecutionElement

**Supertype** ModelElement

**Associations**

- *source*: an association to the DataObject(s) which are the "source" on which the operator is executed.
- *target*: an association to the to the DataObject which is the "target" of the operator execution.

**DataObject** makes the connection between the two parts of the implementation level. It is a placeholder for either a Column, a ColumnSet or a Value which are input and/or output for an ExecutionElement. Only the association to ColumnSet is depicted in the diagram of Figure 9.

### 6.2.2 Transformation

A Transformation may be either a Function, a StoredProcedure or the definition of a SQL-Query (not the result). *Function* and *StoredProcedure* have as attribute *nameOf* which is the name of the function or StoredProcedure that contains the implemented code. *Signature* contains the signature of the called function or stored procedure. *SQL Query* contains (at least) three attributes:
- *select* contains the specification of what has to be selected in the query (i.e., the list of columns),
- *from* specifies the tables or views of which has to be selected
- *where* specifies the conditions for selection
For the operator *Scaling* considered above (see Section 6.1.7), there is at least an instance of the class *StoredProcedure* (let us call it *Scaling* as well). The *StoredProcedure* instance contains only the call of the PL/SQL procedure. The code itself is managed by the database software and look as follows:

PROCEDURE $Scaling(IntervalLowRangeInput$ REAL,
$IntervalHighRangeInput$ REAL, $IntervalLowRangeOutput$ REAL,
$IntervalHighRangeOutput$ REAL,
$NewColumn$ REAL, $OldColumn$ REAL)
IS
$ScalingFactor$ REAL;
BEGIN
$ScalingFactor = (IntervalHighRangeOutput - IntervalLowRangeOutput)$ /
$(IntervalHighRangeInput - IntervalLowRangeInput)$

SET NewColumn AS $IntervalLowRangeOutput +$
$ScalingFactor(OldColumn - IntervalLowRangeInput)$
RETURN;
END Scaling

The *DataObject* instance as "source" of this *ExecutionElement* is the value of *OldColumn* and the "target" is *NewColumn*. Note that Transformations works with implementation data types like REAL, INTEGER and not with (conceptual) mining types (e.g., SCALAR, NOMINAL, BINARY). Moreover, note that the procedure above could be "broken" in smaller code modules, e.g., *Scaling* could call a function *ComputeScalarFactor* which would be then an instance of the class *Function*. In this way, the implementation of operators could be better tracked and maintained.

# 7 Conclusion

This paper presents first ideas of a metamodel design for a metadata-driven software package performing preprocessing for data mining. In the course of software development, metamodel changes have to be expected. However, the main features derived from the four part distinction (data/case modelling vs. conceptual/implementation representation) should be preserved.

Using metadata-driven software is particularly beneficial if enterprise-wide integration of metadata is planned. There are currently two groups proposing metamodel standards to store and exchange metadata within the data warehousing area: Object Management Group (OMG)[10] and Meta Data Coalition (MDC)[11]. The OMG standard CWM (Common Warehouse Metamodel)[7] is restricted to (technical) metadata for data warehousing, whereas the MDC standard OIM (Open Information Model)[6] is much broader in scope, covering, besides data warehousing, also aspects like business engineering (business rules, business processes etc), organizational elements, object-oriented analysis and design etc. A unification of the two standards has been recently announced[12] and will emerge in a new release of CWM. In the future, the metamodel proposed in this paper has to be eventually integrated with the new standard expected to be the unique representation and exchange modality for metadata in data warehousing.

**Acknowledgements:** This paper has emerged from the collaboration between two projects, SMART (Supporting Metadata for Date Warehousing) and Mining Mart (Enabling End-User Data Warehouse Mining). SMART is partly funded by the swiss committee of innovation and technology (project number KTI 3979.1). Mining Mart is a European Commission research project (IST-1999-11993) with support of the Swiss Federal Office for Education and Science (project number BBW 99.0158).

# References

[1] Borgida A. Description logics in data management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):671–682, Oktober 1995.

[10] www.omg.org
[11] http://www.MDCinfo.com/
[12] http://www.cwmforum.org/

[2] R. Brachman and T. Anand. The process of knowledge discovery in database. In *Advances in Knowledge Discovery and Data Mining*. AAAI Press, 1996.

[3] B. Chandrasekaran, J.R. Josephson, and V. R Benjamins. What are ontologies, and why do we need them? *IEEE Intelligent Systems*, 14(1):20–26, January/February 1999.

[4] N. Fridman Noy and C.D. Hafner. The state of the art in ontology design. *AI Magazine*, 18(3):53 – 74, Fall 1997.

[5] J.U. Kietz, R. Zücker, and A. Vaduva. MINING MART: Combining case-based-reasoning and multistrategy learning into a framework for reusing KDD-applications. In *Proc. of the 5th Intl. Workshop on Multistrategy Learning (MSL 2000)*, Guimaraes, Portugal, June 2000.

[6] Microsoft. *Open Information Model*. http://www.microsoft.com/sql/ tech-info/openinfo.htm.

[7] Object Management Group (OMG). *Common Warehouse Metamodel (CWM) Specification*, 2000. OMG Document ad/00-01-01, ad/00-01-02, ad/00-01-03, ad/00-01-11, also see http://www.cwmforum.org/.

[8] D. O'Leary. Using AI in knowledge management: Knowledge bases and ontologies. *IEEE Intelligent Systems*, 13(3):34 – 39, May/June 1998.

[9] D. Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann Publishers, San Francisco, California, 1999.

[10] M. Staudt, A. Vaduva, and T. Vetterli. Metadata management and data warehousing. Technical Report 21, Swiss Life, Information Systems Research, July 1999. ftp://ftp.ifi.unizh.ch/pub/techreports/TR-99/ifi-99.04.pdf.gz.

[11] M. Staudt, A. Vaduva, and T. Vetterli. The role of metadata for data warehousing. Technical Report 99.06., University of Zurich, Dept. of Information Technology, September 1999. ftp://ftp.ifi.unizh.ch/pub/techreports/TR-99/ifi-99.06.ps.gz.

[12] A. Vaduva and K.R. Dittrich. Metadata management for data warehousing: Between vision and reality. Technical report 2000.08, University of Zurich, Dept. of Information Technology, January 2000. ftp://ftp.ifi.unizh.ch/pub/techreports/TR-2000/ifi-2000.08.pdf.

[13] T. Vetterli, A. Vaduva, and M. Staudt. Metadata standards for data warehousing: Open Information Model vs. Common Warehouse Metamodel. *ACM Sigmod Record*, 29(3), September 2000.

# A Appendix

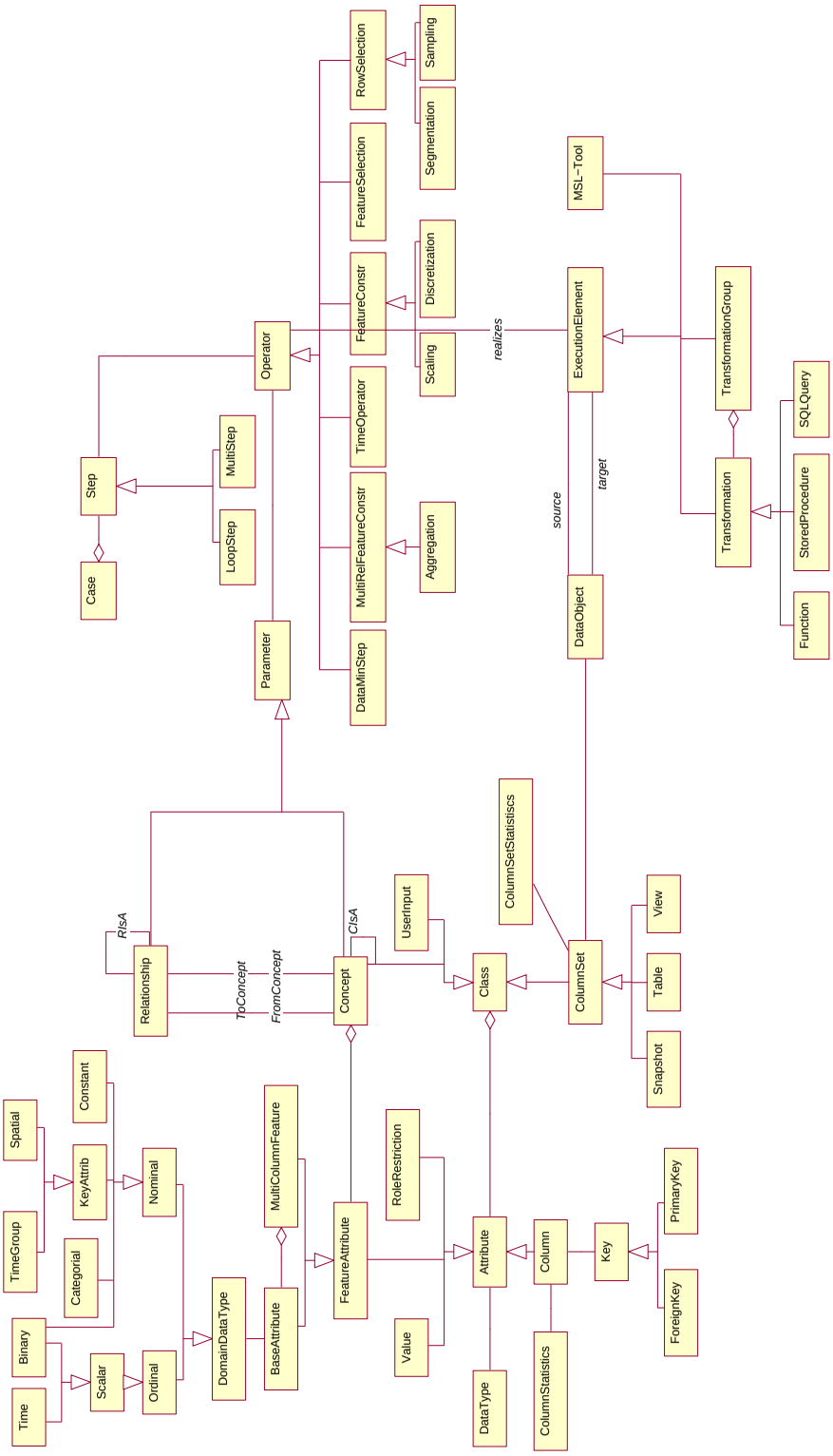


Figure 9: The Mining Mart Metamodel