Let's Modularize the Data Model Specifications of the ObjectLens in VisualWorks/Smalltalk

Michael Prasse

Abstract

The ObjectLens framework of VisualWorks maps objects to tables. This mapping is described in a data mapping model, which itself is specified in one dataModelSpec method. This method is monolithic and defines the whole data model of an application. This is a suitable approach to start with.

However, when the business area extends to a set of similar applications, like a software product family, each of these applications needs its own data model specification. All specifications of the product family would be quite similar but there is no appropriate reuse-mechanism, which could be used. Consequently, the monolithic design specifications lead to a high degree of redundancy, which complicates software development and maintenance.

Therefore, this paper describes an approach, which leads to a separation of the monolithic data model specifications. The main idea is to define the mappings of each class in the class itself using inheritance and generate the whole specification from a list of single class data models. In this way, declarative and generative programming techniques are combined.

Keywords

ObjectLens, Smalltalk, VisualWorks, Design Pattern, Software Product Families, OR-Mapping, Generative Programming

Contents

Introduction

Since 1997 our software engineering group develops applications in the domain of pension schemes with VisualWorks/Smalltalk. At the beginning this software was specified for one customer. In the course of time the number of customers and the application domains grew. Today we support more than 20 customers and all kinds of pension schemes in Germany including long-term accounts of employees. The system architecture is extended from a fat client architecture to an application service provider architecture including a web application server, which is also build in VisualWorks.

To reduce software engineering costs we organized our applications as a product family. There is a single source base for all applications improving reuse of existing modules. The core is organized as a framework including common GUI-standards, common domain specific models, database access layers and standard management and administration modules. The applications extent this core by defining new specific modules using object-oriented techniques like inheritance, object composition and meta programming.

But there is one area where we could not achieve a high degree of reuse directly. This is how object-relational mappings are defined in the ObjectLens framework, which is the heart of the data base access layer. In the ObjectLens, the object-relational mappings are described in one monolithic specification. We need a specific object-relational mapping for each application. All this specifications have common parts. Defining a new specification starts with copying a suitable specification and changing it. This copy-paste approach leads to a high level of redundancy and makes data model changes of common parts more difficult because many specifications have to be updated. In this paper we want to describe a more sophistical solution solving these problems.

The presented solution is a pragmatic one. The first aim was to solve the redundancy and maintenance problem concerning the data model specifications. It was not our goal to do an extensive academic research on object-relational mapping or to develop a new object-relational mapping framework. For example, there is no opportunity to exchange the base object relational mapping of our applications. The costs and time are in no relation to the expected benefits. For these reasons our solution has to be integrated in the existing ObjectLens. Of course to achieve our primary aim of improving the data model specifications we used an engineering approach including analysis, design, risk management, testing and stepwise deployment in production.

The article is structured as following. First the ObjectLens framework of VisualWorks is introduced. Then the data model specifications of the ObjectLens and their disadvantages for our product line approach are discussed in detail. Afterwards we present our solution and its integration in the ObjectLens framework. In the next section we describe the generation of our new data model specification parts form the old specifications and how we solve specification conflicts. The conclusion summarizes our experiences.

ObjectLens Framework

The ObjectLens Framework is an integrated part of VisualWorks since Version 2.0 from 1994 ([Parc 1994], [Cinc 2003a]). The major concepts of the ObjectLens have not changed since then. It is an early developed access framework for mapping objects to relational database tables. It is comparable to other early OR-mapping tools from this time like TOPLink by The Object People Inc., now Oracle, Polar by IBL Ingenieurbüro Letters GmbH, Arcus Relational Database Access Layers by sd&m, MicroDoc Persistence Framework by MicoDoc GmbH or Crossing Chasms Pattern Language by Kyle Brown ([BrWh 1996], [IBL 1998], [KeCo 1996], [Micr 1998], [TheO 1998])[1].

1. We enumerate only approaches which were introduced at the same time like the ObjectLens. Therefore, GLORP (Smalltalk), JDO (Java) or Hibernate (Java, .NET) are not considered ([BaKi 2004], [Knig 2004], [Roos 2003]).

In the next subsections, we describe the architecture, the object mapping to tables and the programming metaphor of the ObjectLens.

Architecture

The ObjectLens Framework consists of four modules, which are described abstractly in figure 1. The declaration module defines the specifications to describe the data model in a logical way. This module contains classes for describing the data model and the data model specification. The data model is a set of objects and defines data structure types, variables, types, and foreign key relationships. The data model specifications are a declarative way to define this data model. Furthermore, it uses the database module, which describes database tables and columns, to specify the mapping to the logical database design. Together, both modules describe the logical database design and mapping. There are also tools like the data modeler and the mapping tool, which allow to specify the data model specifications tool based. Furthermore, you can generate or adapt the logical database structure from a data model specification automatically. This process is called "*check with database*".

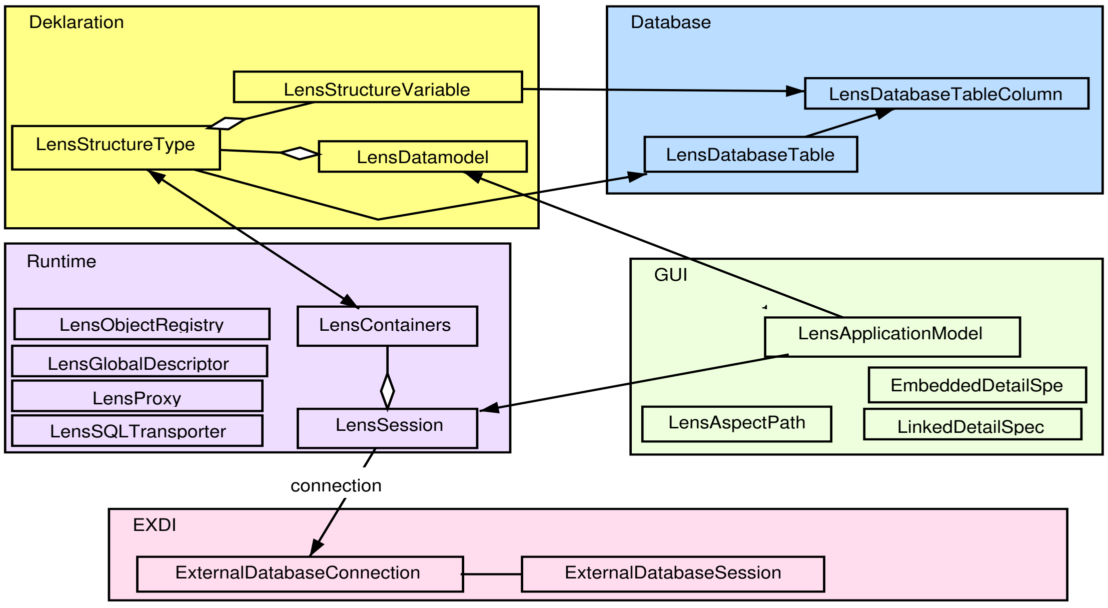


Figure 1: Technical Architecture

The next module is the runtime engine. It defines the infrastructure, which is required for mapping objects to table rows and vice versa at runtime. It contains classes for row containers, caching, proxies, and SQL queries. Furthermore, it defines a lens session, which controls the access to the persistent objects. This module supports a seamless integration of SQL queries into Smalltalk and reduces the impedance mismatch ([CoMa 1984]).

The last module defines GUI widgets for viewing and editing persistent objects. These widgets are integrated seamlessly into the GUI framework of VisualWorks. Transient and persistent objects can therefore be represented in the same way. It defines also aspect paths, which allow connections between the object aspects and the visual components via the ValueModel-pattern ([ABW 1998], [Cinc 2003b], [Howa 1995], [Wool 1994]).

The ObjectLens itself is based on the EXDI framework (external database interface), which provides a low level access to database programming. The EXDI provides a set of abstract protocols to establish a connection to the database server, to prepare and execute SQL queries, to obtain the results, and to disconnect from the server. It supports also flat or non-nested database transactions with begin, commit, and rollback.

The EXDI is an abstract framework. It provides the general implementation, but it does not provide direct support for any particular database. Database Connect extensions are available to provide connectivity to specific databases. Our software engineering group uses database connections for Oracle, DB2, and PostgreSQL. The original ObjectLens framework was build for Oracle and later Sybase. Today we use extensions, which allow to support DB2 and PostgreSQL. Therefore, we can run our applications with different RDBMS without changing any code. The data models of DB2 and PostgreSQL are automatically generated from the data model specification of Oracle. The current database is selected by a configuration file.

Conceptual Mapping from Classes to Tables

The conceptual mapping from classes to tables of the ObjectLens is described in table 1. The ObjectLens uses a simple mapping, which directly maps object-oriented concepts to relational concepts. Classes, instance variables, and monomorphic object references are mapped directly to tables and columns. 1:N and N:M relationships can be modeled by auxiliary classes, which express the relationships or by explicit queries, which select all objects that are connected to the object.

Mapping Support	Concept	Mapping
	calculus level (static semantics)[a]	Φ: object calculus $\rightarrow$ relational calculus
directly	class level	$\Phi_{classes}$: classes $\rightarrow$ tables Each class is unambiguously mapped to one table. In one data model, no table can be mapped to different classes. This restriction holds because the classID of objects is not stored in the tables. Therefore, you cannot map inheritance or polymorphism by storing objects of different classes (subclasses) in the same table.
directly	instance variable level	$\Phi_{instance\ variables}$: variables $\rightarrow$ columns
	instance variables with simple data types	Instance variables with simple data types can be mapped directly to one column.
	instance variables with monomorphic object references (1:1 relationship)	Instance variables that hold object references map to a set of columns, which holds the primary key of this object. This set of columns realizes a foreign key relationship.
indirectly	1:n and n:m relationships	There is no direct representation. Additional tables in the database and select-statements of the ObjectLens can implement these relationships.[b]
no support	inheritance	There is no direct representation. Each subclass is mapped to its own table. Each table contains all instance variables of the class including inherited variables.
no support	polymorphism	Polymorphism for object references is not supported by the ObjectLens. We developed an extension, which allows to support untyped object references. For such references, the foreign key consists of the pair (classID, objectID).

Table 1: Object Relational Mapping of the ObjectLens

a. For an introduction to relational databases and the relational calculus see [Date 1995] or [ElNa 1989]. For an introduction to object calculi and object-oriented concepts see [AbCa 1996], [Meye 1997] or [Prass 2002].

b. ObjectLens select-statements are Smalltalk statements, which are automatically transformed in SQL queries by the ObjectLens.

Unfortunately, the ObjectLens has only restricted support for inheritance and polymorphism. Each class is always unambiguously mapped to one table. You cannot map several classes to one table. Therefore, you cannot map a class hierarchy to one table. That means that you need several queries if you want to select objects of different subclasses. Alternatively, you can replace inheritance by object composition but this can lead to a complicated class design.

For example, you could use a class *A* for the queries, which has no subclasses. If class *A* is used for queries, then only one query is needed to access all objects. The class *A* has a reference to the root class *B* of a class hierarchy. This reference is mapped by an untyped object reference so that objects of *A* can point to objects of subclasses of *B*. The cost of this design is the separation of one domain entity in two subparts.

Foreign key relationships can only be mapped for monomorphic instance variables. That means such a variable can only hold objects from one class. If you want to use polymorphic variables, which can reference to objects from different classes, you have to build untyped relationships. However, in this case you have to manage the access itself.

Programming Metaphor

The ObjectLens uses explicit persistency. The metaphor of persistency of the ObjectLens is the persistent container or collection. It uses no persistency by reachability, where all objects, which are reachable from a persistent root object are persistent, or persistent classes, where all objects of the class are automatically persistent.

The ObjectLens is interpreted as a collection. To make an object persistent, it is simply added to the lens session. To remove an object from the database, it is simple removed from the lens session. The syntax is comparable to theirs of collections:

- to make an object persistent: *aLensSession add: anObject*
- to remove an object from the database: *aLensSession remove: anObject*

For all objects, which are included in the lens session, changes are automatically detected. Each state change of such an object leads to an *isDirty*-registration. This *isDirty*-mechanism is integrated in the setter methods of all instance variables by using the private method *update:to:* of the ObjectLens. Each state change using a setter method is therefore detected. To reduce coding errors, the getter and setter methods for instance variables can be automatically generated.

The ObjectLens supports flat transactions. Therefore, all updates, which occur in a transaction, are written either together in the database (commit) or are rejected (rollback). Furthermore, you can use the ObjectLens without transactions. In this case changes are immediately written into the database.

Database queries can also be written in Smalltalk. The syntax is comparable to the method *select* of collections. The base for queries is the class LensQuery. Where-clauses are expressed as block closures like in the collection methods *do:*, *select:* or *detect:*. Figure 2 shows an example of a select-statement from our domain.

```
readEmployees: anEmployer in: anApplication
    ^anApplication
        selectOnContainer: self container
        whereBlock: [:each | each employer = anEmployer & each isCurrent]
```

Figure 2: Select-statement in Smalltalk

The result list of a query is automatically transformed into corresponding objects. Object references are expressed as lens proxies. If a proxy is accessed it is automatically resolved by the corresponding object. An object cache ensures referential integrity. All this mechanisms help to abstract from the relational persistent mechanism and the database access in the ObjectLens. In most cases, Smalltalk syntax can be used for persistent objects, which reduces the impedance mismatch.

The ObjectLens supports multiple lens sessions. An application can use several lens sessions to access different databases simultaneously. However, the ObjectLens has no multi-process ability. It is impossible to access one lens session from different threads or processes. The implementation of the ObjectLens uses singletons for building the SQL-requests and is therefore not thread safe.

One further disadvantage is the bad performance by mass queries, if object references are resolved by single queries. This is a typical trade-off of object references and object navigation. Object navigation is fast, but a query over a set of such objects needs additional queries for each object in the set. You can influence this by using explicit select statements for mass queries.

Summary

The ObjectLens together with the EXDI framework and the specific Database Connect extensions provides support for the most relevant aspects of building database applications. This includes the declaration of the mappings, the creation and adaptation of the database tables, the low level database access, the creation of user interfaces for persistent objects and the runtime support with storing objects into the database, retrieving objects from the database and querying the database, which Smalltalk queries, which are translated into SQL.

The ObjectLens provides also a simple mapping from object-oriented concepts to relational concepts. Inheritance and polymorphism are not directly supported. Nevertheless, there are ways to achieve both.

In the most cases you can think about the ObjectLens as a persistent collection. To make an object persistent, you add it. To remove an object from the database, you remove it from the Lens. To select an object from the database, you send a select statement to the Lens. The technical aspects like transactions, proxies, posting updates, and translating queries into SQL are done by the ObjectLens.

Summarizing, the ObjectLens is an object-oriented access layer to relational databases. Its advantages are:

- a seamless integration in VisualWorks
- good support access by navigation and single queries
- the generation of the database scheme
- RDBMS-abstraction (Oracle, Sybase, DB2, ODBC, (PostgreSQL))
- GUI-support
- the support of multiple lens sessions
- graphical modeling tools for describing and generating data models

Its disadvantages are:

- only rudimentary support of inheritance and polymorphism
- bad performance by mass queries
- no multiprocessor ability

Data Model Specification

After introducing the ObjectLens let us look now at the data model specifications, which describe the mapping for one application. In the next subsections, we describe the structure of the *dataModelSpec*, the problems of maintenance and first solutions.

Conceptualization

The *datamodelSpec* is a declarative description of a lens data model. It is coded as a literal array (*LiteralArray*). A literal array is an array of arrays of literals. It is recursively defined. Literal arrays are widely used in VisualWorks. Its most prominent use in VisualWorks is the *windowSpec* of the GUI-Framework. All windows, which use the VisualWorks framework, are declarative described by literal arrays. Another example is the specification of diagrams by the Advance UML modeling tool of VisualWorks, which uses *ad2diagram* methods ([Cinc 2003b], [Howa 1995]).

To encode a lens data model, you use the method *literalArrayEncoding*. To decode a lens data model, you use the method *fromLiteralArrayEncoding:*.

- encoding: *aLensDataModel literalArrayEncoding* returns a literal array suitable for reconstituting the receiver.
- decoding: *LensDataModel fromLiteralArrayEncoding: anArray* creates a lens data model from the array encoding.
- *LensDataModel fromLiteralArrayEncoding: (aLensDataModel literalArrayEncoding)* returns a lens data model, which is equal to *aLensDataModel*.

If you apply the methods *literalArrayEncoding* and *fromLiteralArrayEncoding:* alternately then you can switch between the data model level and the data model specification level. This means, that you can choose the language level for the specification of the data model specifications.

Figure 3 shows the general structure of a literal array and of a *dataModelSpec* method. A literal array consists of two central parts: a class and a set of (aspect, value) pairs. The class determines the kind of object, which the literal array describes. The (aspect, value) pairs describe the state of the object. Usually the aspect is a method selector and the value is the argument. The receiver of the object is the recently constructed object. In general, the construction process uses therefore a set of method sends of the form '<object> <aspect> <value>'. The value itself can be encoded as a literal array leading to nested encodings.

literal array
```
    ^#(<Class>
        <aspect> <value> <aspect> <value> <aspect> <value> ...)
```

lens literal array
```
^#(#{Lens.LensDataModel}
        #setDatabaseContext: #(...)
        #structureTypes: #(
                #(#{Lens.LensStructureType}
                                #memberClass: <memberClass>
                                #setVariables: #(
                                    #(#{Lens.LensStructureVariable}
                                            #name: 'angelegtAm'
                                            #column:  <Database Column>
                                            #privateIsMapped: true )
                                        ...)
                                #table: <Database Table> )
                    ...)
        #lensPolicyName: #Mixed
        #lensTransactionPolicyName: #PessimisticRR
        #validity: #installed )
```

Figure 3: Structure of Literal Array and Lens Literal Arrays

The general structure of a lens literal array is also described by figure 3[1]. The first aspect defines the database context. The second aspect describes the containing structure types. This is the most important aspect of the description because a lens data model is mostly a set of structure types. The next two aspects describe policies. The validity aspect determines the definition state of the data model.

The literal array of a lens structure type determines the class of the structure type, the variables of the structure type and the table. The literal array of a lens structure variable determines name, mapping, and column of the variable. The <value> for the aspect #strutureTypes: is a collection of structure types and the <value> of the aspect #setVariables: is a collection of structure variables.

Figure 4 shows the beginning of an existing *dataModelSpec* method, which is part of our system. As shown in figure 3 the structure types are described as literal arrays. The definition database is an *Oracle7Context* with user name 'lens' and database 'lensDB'. The example shows the beginning of the specification of the lens structure type *COLAdresse*. A lens structure type itself consists of a set of lens structure variables. In the example, the definition for the variable 'dependents' is shown. This is an unmapped (transient) variable. The *datamodelSpec* can specify persistent variables, which are mapped, and transient variables, which are unmapped. Each lens structure type, which is used as a type of a lens structure variable, has to be defined in the *dataModelSpec*. This is a completeness constraint to the specification.

```
dataModelSpec
    "LensEditor new openOnClass: self andSelector: #dataModelSpec"

    <resource: #dataModel>
    ^#(#{Lens.LensDataModel}
        #setDatabaseContext:
        #(#{Oracle7Context}
            #username: 'lens'
            #environment: 'lensDB' )
        #structureTypes: #(
            #(#{Lens.LensStructureType}
                #memberClass: #{COLAdresse}
                #setVariables: #(
                    #(#{Lens.LensStructureVariable}
                        #name: 'dependents'
                        #setValueType: #Object
                        #generatesAccessor: false
                        #generatesMutator: false
                        #privateIsMapped: false ) ...
```

Figure 4: Example dataModelSpec

Maintenance Problems

The maintenance problems, which we identified by using the ObjectLens, can be classified into two groups. The first group contains problems, which result from the poor support of inheritance by the ObjectLens. In the second group are problems, which result from the move to a product family with different *datamodelSpecs*. The origin of both problem groups is redundancy.

Class Hierarchy Problems

The lens structure type of a class defines all instance variables of a class including inherited variables. This is necessary because the corresponding table has to store all variables of the objects. Therefore, in each subclass of a class all instance variables of that class have to be defined once again. There is no single source principle for the specification of the mapping of instance variables.

These multiple definitions lead to some maintenance problems. If a new subclass is added to a *dataModelSpec* using the lens data modeler, the mappings of the inherited instance variables are not

1. The literal array is not described in all details. Only the most important aspects are shown.

taken over. They have to be specified again, what is cumbersome and error prone. If a new variable is added to a superclass then all lens structure types of the subclasses have to be changed. The renaming of an instance variable of a superclass requires analogous adaptations. The same instance variable can be mapped variously to the database in different subclasses. In some situations, this flexibility could be an advantage. More often, different mappings are unwanted and only the result of missed adaptations. For example the property sex of a person is mapped to {'m','f'} in some subclasses and to a boolean in other subclasses of the same data model.

The same situation occurs if the superclass of a class is changed. In this case all instances variables of the old superclass have to be removed from the specification and all instance variables of the new superclass have to be added with the correct mapping. Figure 5 shows an example. The superclass of the class *AtzBeleg* is changed from *ZEBeleg* to *BelegMitRechtskreis*. The red-colored variables are changed. They need therefore a new mapping. If you remember that these are information of the superclass you understand that each change in the class hierarchy inflicts subclasses directly.

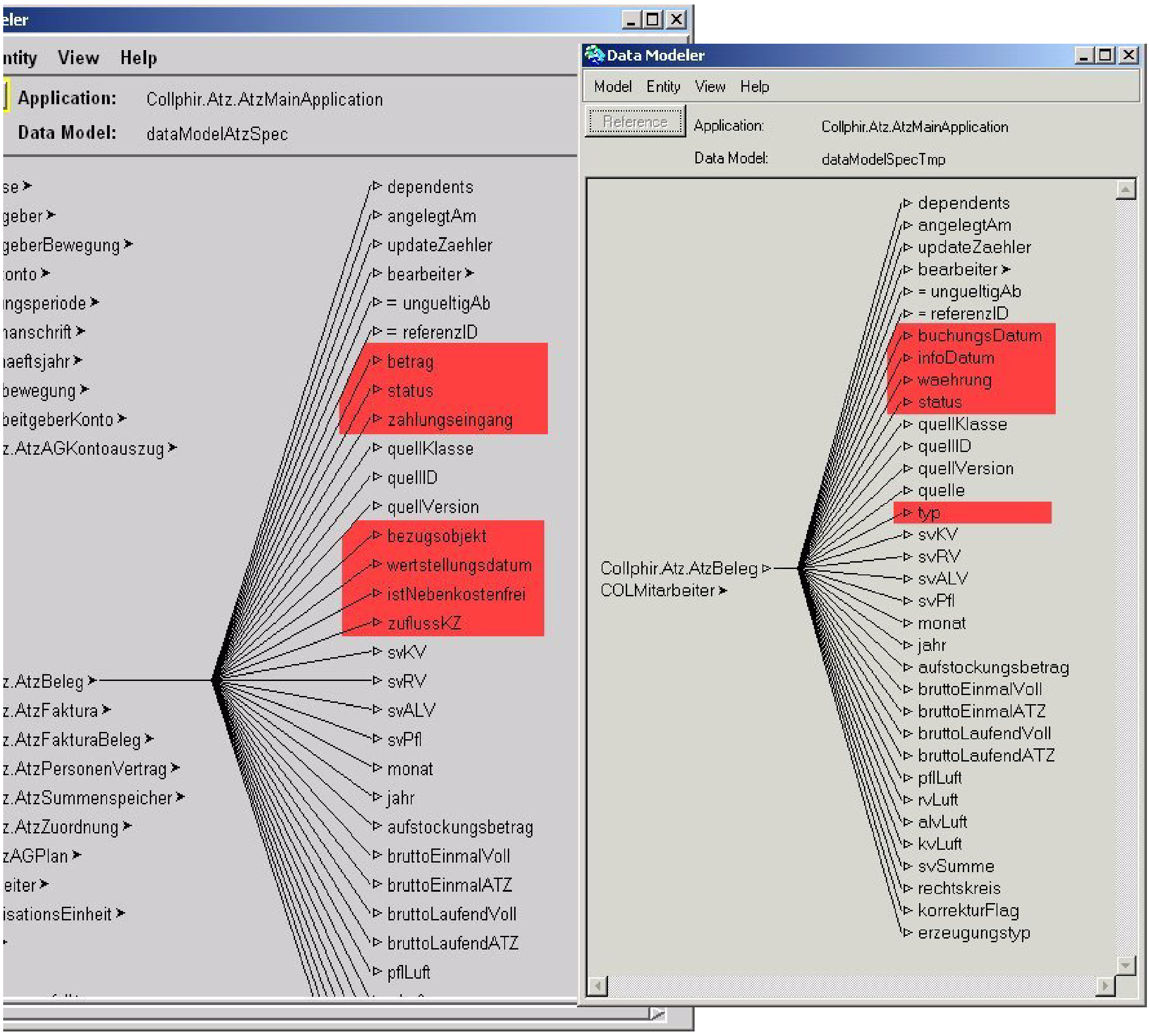


Figure 5: Changing the Superclass

Multiple datamodelSpec Problems

One monolithic *datamodelSpec* is used to describe the data model of an application. The data model has to contain all entity classes of the application. When we switched from one application to a product family, we had suddenly to deal with multiple *datamodelSpecs*. Common shared core modules and domain specific modules characterize the product family. If we start a new project for a customer in the domain context, we often copy and paste an existing *dataModelSpec* of an old project. Then this *datamodelSpec* is adapted to the new requirements.

Concerning the common core modules, all our *dataModelSpec* have overlapping parts. So changing the mapping of a superclass in a core module results not only in modifications of the subclasses in one *datamodelSpec*, but also in all the other *datamodelSpecs*. Extending a core module by a new persistent class requires again modifications to all *datamodelSpec*. If these changes are not maintained to all applications then inconsistencies and different mappings may arise.

The origin of all these problems is the redundant specification of instance variable mappings in subclasses and *datamodelSpecs*. There is no single source principle for specifications of the ObjectLens.

The DataModelMerger as a first Solution Approach

The main idea to resolve the maintenance problem is to reduce the redundancy. Our first approach was to separate the *datamodelSpec* into different parts. We specified complete *datamodelSpecs* of sub domains. These *subdatamodelSpecs* are comparable to *subcanvasSpecs* of the GUI framework and are merged into one *dataModelSpec* by a data model merger (figure 6).

```
DataModelMerger new
        mergeAll: (OrderedCollection new
                        add: self dataModelVifaSpec;
                        add: self dataModelAtzSpec;
                        add: self dataModelSVLuftSpec;
                        yourself)
        ignore: #(#(
                           #COLAZ03)
                           #(#COLAZRR)
                           #(#COLRueckzahlungssatz) )
```

Figure 6: Data Model Merger

The composition of the *subdataModelSpecs* is simple. All structure types of the *subdataModelSpecs* are added to the aspect 'structueTypes' of the composed *dataModelSpec* (see figure 3 for the base structure of a lens data model). If a structure type is already included in the composed data model then another structure type of the same *memberClass* is not added once again. In this way, a coarse-grained modularization of the ObjectLens is achieved.

Nevertheless, this approach remains unsatisfactorily. At first, it solves not the class hierarchy problems. At second, the domain *subdataModelSpecs* are still too extensively. Each *subdataModelSpec* has to be complete with regard to all used lens structure types. Therefore, there are common classes like *Employee*, *Employer*, or *Person*, which are included in all *subdataModelSpecs*. At third, the *subdataModelSpecs* includes often classes, which are not needed. Some of these classes can be removed from the data model by including these into the ignore set.

Summary

In conclusion, the *dataModelSpec* is a declarative description of a data model. It is coded as a literal array. Unfortunately, the *dataModelSpec* is a monolithic definition, which has only limited support for inheritance. Therefore, a number of problems occur during defining and maintaining such data model specifications. The origin of all these problems is the redundant specification of instance variable mappings in subclasses and *datamodelSpecs*. This redundant definition leads to problems when adding or changing variables of a superclass, when adding a new subclass or when changing the superclass. Specification conflicts can occur if the same variable is mapped differently in different subclasses. The main idea to resolve these maintenance problems is to reduce the redundancy.

Modularization of the ObjectLens

Up to now, we introduced the ObjectLens and described their relational database mapping and associated disadvantages. Now we explain our approach to overcome these problems in the following sections. At first, we describe the general ideas and aims of the solution. Then we point out the definitions of the lens mapping for the single domain classes. After that, we demonstrate the integration of the mappings of the single domain classes into one data model of the lens application. Then we explain the migration of our old data models into the modular data models. At the end, we show the integration of our approach into the common database developer tools of VisualWorks.

General Ideas and Goals

The general aspects of our solution are modularization and the use of inheritance. If you remember, the lack of inheritance and the monolithic design of the data model specifications of the ObjectLens are the origin of redundancy and the related problems. We decided to break up the monolithic specification in several pieces with each piece describing the mapping of one class[1]. Furthermore, we use inheritance if we want to describe the object relational mapping for one class. Therefore, only the parts of a class without inherited variables have to be considered. The data model specification of a lens application is defined by the data model specifications of the contained set of classes. That means that the single class data specifications are the pieces from which the whole data model specification is constructed. The result is a normal, but generated monolithic data model specification of the ObjectLens. Therefore, we changed only the definition and construction process of the data model specifications.

This approach gives us the desired advantages. We achieve a better adaptation, a unification of the data representation of different applications in the product family and the use of inheritance. In some way, we look at the *datamodelSpec* as one aspect of the class and organize this aspect by the class itself. The modularization of the *datamodelSpec* simplifies the maintenance affords significantly. Instead of changing a central monolithic definition, we change only the modular definitions of the concerned classes.

Therefore, our solution consists of four parts. We store the mappings in the domain classes. We construct automatically the *datamodelSpec* from these mapping fragments. We support the common development tools. We support the migration of our existing data model specifications.

Data Model Mappings of Classes

The data model specification of a class defines the corresponding lens structure type, whereby definitions of inherited variables are obtained from the superclasses. The definition of one class uses the definition of the superclass. Variables are described as lens structure variables (remember figure 3). The lens structure type of a single class can easily be integrated in the aggregated data model specification.

Figure 7 shows the public protocol for defining the data models of a class. This definition uses the template method pattern like the methods *printString* and *printOn*: ([ABW 1998], [GHJ+ 1998]). The method *dataModelDefinition* provides an abstract implementation, which should be used by all classes. The method *dataModelDefinition* should not be overridden. First, the method *primDataModelDefinition* is called, which provides the standard implementation. After that, the method *primLocalDataModelDefinitionChanges*: is called. This method gives each class the opportunity to override the inherited definitions. Whereas the method *primDataModelDefinition* will usually be automatically generated, the method *primLocalDataModelDefinitionChanges* is created by hand and describes changes, which should not be overridden by further generation steps. The persistent classes of our product family are

1. This approach is comparable to instVarMaps of GemStone. You can control instance variable mapping between GemStone and your client Smalltalk by using these methods ([Gems 1996]).

```
dataModelDefinitionSpec
    " You should not override this message. "
    ^ self dataModelDefinition literalArrayEncoding

dataModelDefinition
    " You should not override this message. You can adapt primDataModelDefinition"

    | type |
    type := self primDataModelDefinition.
    self primLocalDataModelDefinitionChanges: type.
    type variables: (List withAll: type variables ).
    type resolveStandalone.
    ^type
```

Figure 7: Public Protocol for Class Data Model Definitions

subclasses of *COLPersistentModel*. Therefore, we define the template methods for defining the lens structure types in this class in the method protocol '*lens data model specs*'.

Figure 8 shows the basis hook method of the class *COLPersistentModel* and a further example. It also displays the usual way in which a lens structure type is defined. We use the *LensMetaData* classes directly. At first, we create an object of class *LensStructureType*. After that, the member class and the table are set up. The other example demonstrates the definition of structure variables of the persistent instance variables. Here we use the literal encodings. The decision to use literal encodings for variables is a pragmatic one. We want to simplify the migration process of our existing *dataModelSpecs* and we want to use the facilities of the ObjectLens for generating lens encodings. Variables with simple data types are directly included in the method. Instance variables for object references (foreign key relationships) are defined in separate methods, because our objects use two-dimensional primary keys and therefore the corresponding literal encodings are more complex. At the end, primary key and table name are defined[1].

```
COLPersistentModel>>primDataModelDefinition
    "hook method"

    | type |
    type := LensStructureType new.
    type memberClass: self.
    type table: ((Oracle7Table new) name: self name; owner: 'COLBAV').
    type idGeneratorType: #userDefinedId.
    ^type

 primDataModelDefinition
    | type |
    type := super primDataModelDefinition.

    type variables add: #(#{Lens.LensStructureVariable} #name: 'name' #setValueType: #String #fieldType:
#String #column: #(#{Oracle7TableColumn} #name: 'name' #dataType: 'varchar2' #maxColumnConstraint: 100)
#generatesAccessor: false #generatesMutator: false #privateIsMapped: true) decodeAsLiteralArray.

    self addSummenspeicherVariableIn: type.

    type idVariable: #('ungueltigAb' 'referenzID') .
    type table name: 'kontoZuordnung' .

    ^type
```

Figure 8: Hook Method primDataModelDefinition

1. In general the primary key is taken from the superclass and the table name is set to the name of the class.

Figure 9 shows an example for the hook method *primLocalDataModelDefinitionChanges*:, which can be used for adapting inherited properties. In the example, the variable *speicherBeleg* get a new type. On the database the variable *speicherbeleg* is mapped as a foreign key relationship to the table of *AtzSummenspeicherBeleg*. This allows the simulation of covariant instance variable redefinitions[1].

```
primLocalDataModelDefinitionChanges:type
    | var |
    super primLocalDataModelDefinitionChanges:type.
    (type variableNamed: 'speicherBeleg') setValueType: #AtzSummenspeicherBeleg.
```

Figure 9: Hook Method primLocalDataModelDefinitionChanges

The hook methods *primDataModelDefinition* and *primLocalDataModelDefinitionChanges* are used to define a lens structure type of a class. The template methods *dataModelDefinition* and *dataModel-DefinitionSpec* are the public interface. They are used for integrating the class fragments into the whole data model specification.

LensApplication datamodelSpec

Now we consider the application side. Like we showed above, the old data model specification describes the data models of the persistent classes of an application. Therefore, we need to define the set of classes, which belong to the data model. This is done by the class method *dataModelClasses*. The set has to include all classes, which are referred in the data model (transient closure), otherwise the data model specification cannot be created. We choose this decision to make the declaration explicit. There are methods, which can calculate the transient closure of a set of classes so that the resulting data model is complete.

The second step is the generation of the whole data model specification from the data model classes. We describe this construction top down. The top method is the method *dataModelSpecGenerated* (figure 10). In this method, an object of *LensDataModel* is created from the specifications of the data model classes. This is done by the code fragment *"self dataModelSpecForStructureTypeSpecs: self dataModelStructureTypeSpecs"*. The method *adaptDataModel* is a further hook method, which permits of adaptations, which are only valid for this special application. In the last step, the data model is compiled and the method returns the literal encoding of the data model. This method is quite short in contrast to our old *dataModelSpecs* with more than 15000 LOC of formatted code.

```
dataModelSpecGenerated
    | ldm |
    (ldm := LensDataModel new)
        application: self;
        fromLiteralArrayEncoding: (self dataModelSpecForStructureTypeSpecs: self dataModelStructureTypeSpecs).
    self adaptDataModel: ldm.
    ldm compile.
    ^ldm literalArrayEncoding
```

Figure 10: LensMainApplication class >> dataModelSpecGenerated (Part 1)

Now we consider the method *dataModelSpecForStructureTypeSpecs* and its implementation (figure 11). The method returns the data model specifications of the data model classes. The method *data-ModelStructureTypeSpecsFor*: shows the connection to the data model specifications of the classes (see Data Model Mappings of Classes, S. 12). For each class in the set of data model classes the corresponding literal encoding is collected.

1. For an explanation of the co- and contravariance issue of object-orientation see [AbCa 1996], [CHC 1990], [Prass 2002].

```
dataModelStructureTypeSpecs
^ self dataModelStructureTypeSpecsFor: self dataModelClasses

 dataModelStructureTypeSpecsFor: classColl
^ (classColl collect:[ :cl | cl dataModelDefinitionSpec]) asArray
```

Figure 11: Methods dataModelStructureTypeSpecs and dataModelStructureTypeSpecsFor: (Part 2)

The last step concerns the implementation of the method *dataModelSpecForStructureTypeSpecs* (figure 12). The array of data model specification literal encodings for the data model classes is inserted in the data model template. The method *dataModelTemplate* provides the general template of the lens data model encoding (see also figure 3). The array of structure types is put at position 5.

```
dataModelSpecForStructureTypeSpecs: aColl
    | res |
    res := self dataModelTemplate copy.
    res at: 5 put: aColl.
    ^res

dataModelTemplate
    ^#(#{Lens.LensDataModel}
        #setDatabaseContext:
        #(#{Oracle7Context} ... )
        #structureTypes: #()
        #lensPolicyName: #Mixed
        #lensTransactionPolicyName: #PessimisticRR
        #validity: #installed )
```

Figure 12: Method dataModelSpecForStructureTypeSpecs: and dataModelTemplate (Part 3)

These few methods describe the generation of the data model specification of the application from the specification fragments of the data model classes. The two central aspects are the determination of the set of data classes and the knowledge that for the generation of the data model specification of the application only the specifications of the lens structure types have to be inserted.

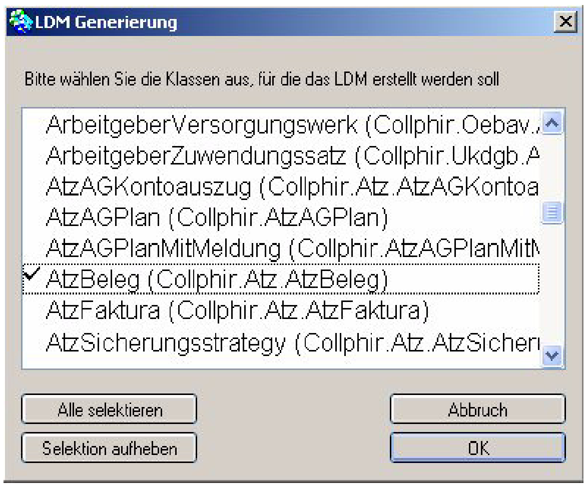
Integration into the Lens Modeling Tools

Now, the integration into the lens modeling tools is explained. One of our goals was to support the lens modeling tools so that each developer can use these tools in the usual way. Otherwise, the acceptance of the new approach would only be low.[1]

The first tool, which we want to support, is the lens editor. The lens editor shows the classes of a data model. Therefore, we provide an opportunity to generate a lens data model for a single class or a set of classes. This is shown in figure 13. We extend the lens editor by a further selection dialog, which allows the selection of data classes. The method *openLensEditorFor:with:* is called for the set of selected classes. In the example, only class *AtzBeleg* is chosen. The method *openLensEditorFor:with:* calculates all classes, which are needed to construct a complete data model. Therefore, the data model contains not only class *AtzBeleg* but also further classes, which are referred by *AtzBeleg*. The so generated data model can be manipulated in the same way like the old data models.

Secondly, we support the mapping tool. The mapping tool allows the definition of the mapping between variable and column. In the mapping tool only a single class is considered (figure 14). Therefore, the mapping tool is the suitable place for creating the class data model. We integrated a new menu item *'Generate Lens Mapping for Class...'*, which opens a multi-selection dialog for the class and its

1. The development of new lens tools was beyond the scope of our solution.

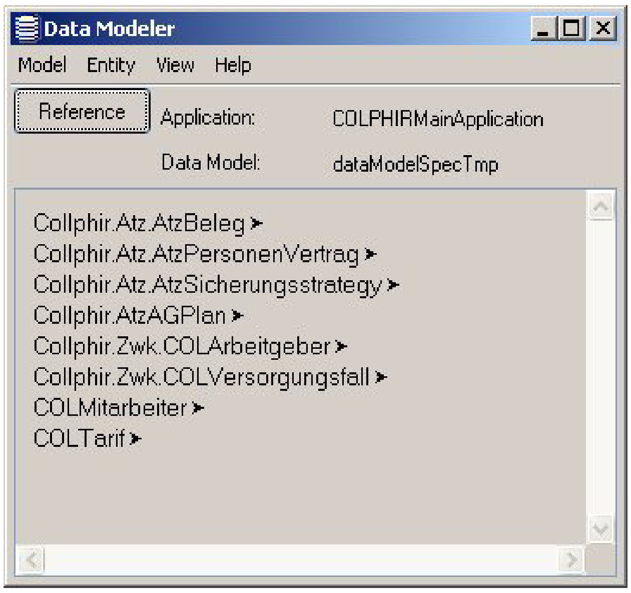


LDM Generierung

Bitte wählen Sie die Klassen aus, für die das LDM erstellt werden soll

- ArbeitgeberVersorgungswerk (Collphir.Oebav.
- ArbeitgeberZuwendungssatz (Collphir.Ukdgb.A
- AtzAGKontoauszug (Collphir.Atz.AtzAGKontoa
- AtzAGPlan (Collphir.AtzAGPlan)
- AtzAGPlanMitMeldung (Collphir.AtzAGPlanMit
- ✓ AtzBeleg (Collphir.Atz.AtzBeleg)
- AtzFaktura (Collphir.Atz.AtzFaktura)
- AtzSicherungsstrategy (Collphir.Atz.AtzSicher

```
LensMainApplication
    openLensEditorFor: CollphirMainApplication
    with: (Set new add: AtzBeleg; yourself)
```



Data Modeler

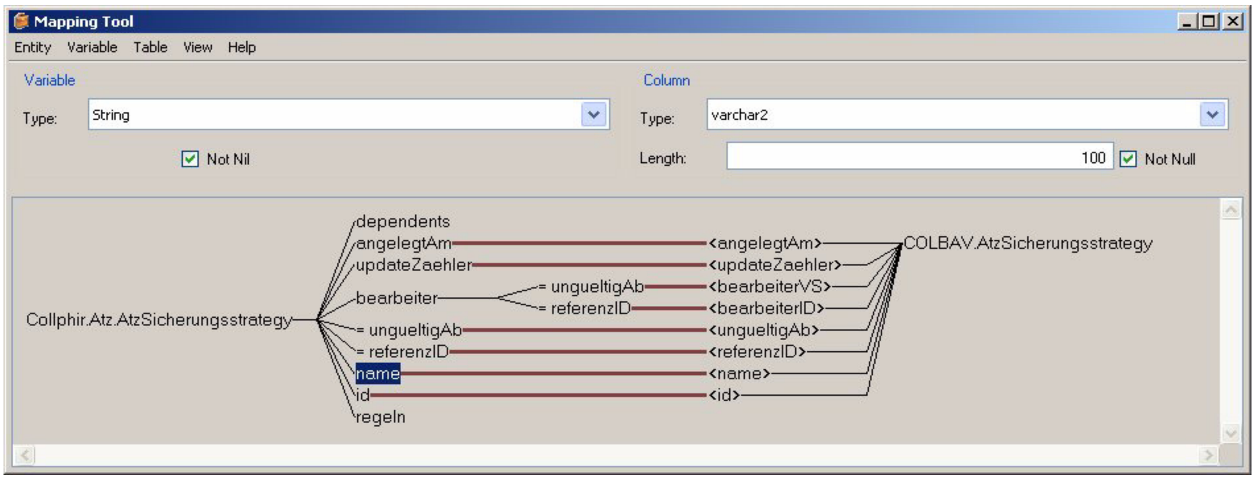
Application: COLPHIRMainApplication

Data Model: dataModelSpecTmp

- Collphir.Atz.AtzBeleg ➤
- Collphir.Atz.AtzPersonenVertrag ➤
- Collphir.Atz.AtzSicherungsstrategy ➤
- Collphir.AtzAGPlan ➤
- Collphir.Zwk.COLArbeitgeber ➤
- Collphir.Zwk.COLVersorgungsfall ➤
- COLMitarbeiter ➤
- COLTarif ➤

Figure 13: LensEditor for Class Data Models

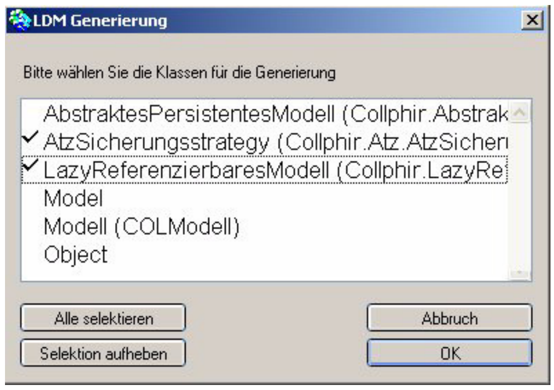
superclasses. The *DataModelDefinitionGenerator* generates the method *primDataModelDefinition* for the selected classes. Remember, the method *primLocalDataModelDefinitionChanges* is not generated.



Mapping Tool

Variable — Type: String; ✓ Not Nil

Column — Type: varchar2; Length: 100; ✓ Not Null



LDM Generierung

Bitte wählen Sie die Klassen für die Generierung

- AbstraktesPersistentesModell (Collphir.Abstrak
- ✓ AtzSicherungsstrategy (Collphir.Atz.AtzSicher
- ✓ LazyReferenzierbaresModell (Collphir.LazyRe
- Model
- Modell (COLModell)
- Object

```
DataModelDefinitionGenerator new
    generateLensSpecsFrom: self ldm
    for: selectedClasses
```

Figure 14: Lens Mapping Tool for Class Data Models

DataModelDefinitionGenerator - Generation and Migration

The class *DataModelDefinitionGenerator* is responsible for the generation of data model fragments. We use the *DataModelDefinitionGenerator* for migration of old monolithic *dataModelSpecs* as well as for generating class data models in the mapping tool.

```
I). transformation T
    generator := DataModelDefinitionGenerator new
        add: AtzMainApplication dataSpec: #dataModelSpec;
        add: ZwkMainApplication dataSpec: #dataModelSpec;
        yourself.

II). conflict reports
    generator report

III).data model classes
    generator
        generateDataModelClassesFor: AtzMainApplication dataSpec: #dataModelSpec .

IV).generating of all classes
    generator generate

    generating of a subset of classes
    generator generateLensSpecsFrom: ldm
            for: (Set new add: COLRente;add: COLAZ03 ;add: COLAZRR ;yourself)
```

Figure 15: Migration Process

The major steps of the migration process are described in figure 15. The *DataModelDefinitionGenerator* can transform a set of data models into a nested dictionary structure. This structure is described by the transformation function *T*. The semantic domains are named after their corresponding classes:

$T\!: IP(LensDataModel) \rightarrow Dictionary[Class, Dictionary[Symbol, Collection]]$ *with:*

- $IP(X)$ *is the power set of* X *with:* $IP(X) =_{df} \{S\!: S \subseteq X\}$

- $aDictionary =_{df} \{key_i \rightarrow value_i : i = 1..n\}$

- $T(\{aLensDataModel_i : i = 1..n\}) =_{df} \{cl \rightarrow aDictionary_{cl} :$

  $\exists k\, \exists s\, (s <_i cl \wedge$

  $LensStructureType_s \in aLensDataModel_k \wedge k \in \{1, .., n\})^1\}$

- $aDictionary_{cl} =_{df} \{\#type \rightarrow Set[LensStructureType],$

  $\#variables \rightarrow aDictionary_{cl,\, variables}\}$

- $aDictionary_{cl,\, variables} =_{df} \{symbol \rightarrow Set[LensStructureVariable] :$

  symbol is a name of a instance variable, which is defined in the class cl}

For each class the structure of dictionaries collects a set of corresponding lens structure types and for each instance variable a set of corresponding lens structure variables. Furthermore, the dictionary structure includes all superclasses and their instance variables. The cardinality of the set of lens structure types and of the set of lens structure variables counts the number of definitions and is a measure of the degree of redundancy. The transformation *T* collects all definitions for a single mapping and merges all considered data models into one single structure.

Figure 16 illustrates the transformation and shows a simplified object view of transformation *T*[2]. The dictionaries cluster and order the information hierarchical. The hierarchy-levels are determined by the structure of a lens data model. The essential information is in the leaves of this tree. The class is associated with its lens structure types. Each instance variable is associated with its lens structure vari-

1. There exists a number *k* and a subclass *s* with the following property: The class *s* is a subclass of *cl* and a *LensStructureType* for class *s* is a member of the *LensDatamodel* with number *k*.
2. We use a simplified notation that is inspired by the object diagrams of UML ([RJB 1999]).

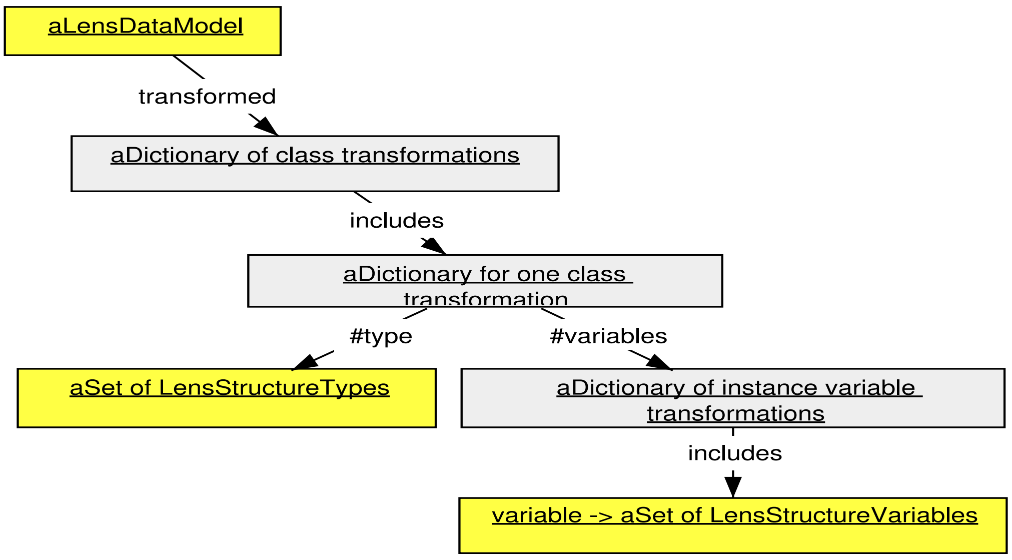

aLensDataModel
transformed
aDictionary of class transformations
includes
aDictionary for one class transformation
#type
#variables
aSet of LensStructureTypes
aDictionary of instance variable transformations
includes
variable -> aSet of LensStructureVariables

Figure 16: Simplified Object View of Transformation T

ables. These lens structure variables are collected from all subclasses of the class, which occur in the data models.

In the following step, we calculated the conflicts between the different definitions of an entity. Here, conflicts during the migration process were handled by a two-step strategy. At first, we eliminated trivial conflict cases and tried to resolve as much conflicts as possible. For example, if different max column constraints occur, then we chose often the weakest one. Then we used pair reviews and decided, which mapping should be become the standard. In a second step, we supported different mappings by using the methods *primLocalDataModelDefinitionChanges* and *adaptDataModel*, which allows overriding already generated properties.

After that, we generate the code in two steps. First, we generate the method *dataModelClasses* for the application. Then we generate the *primDataModelDefinition* method from the corresponding *dictionary* $_{cl}$. The method *primDataModelDefinition* is an aggregation of all lens structure variables of the instance variables, which are defined in this class. Therefore, the method includes literal encodings for each self-defined instance variable. Simple data mappings are inlined. Complicated mappings for foreign key relationships are extracted in separate methods.

For the code generation itself we use common Smalltalk techniques. We defined methods for invariant code fragments and methods, which provides a string representation for related parts of the mapping like table name, primary key, or variables. Then we used a stream to merge this fragments. The result is the source string of a Smalltalk method that we compiled in the metaclass of the considered class in the protocol *'lens data model specs'*.

Summary

The general aspects of our solution are modularization and the use of inheritance. The modularization of the ObjectLens was a four-step process. Firstly, we defined the structure of the specifications of the single data classes. Each data class got a description for its lens structure type. Secondly, we defined the generation of the data model specification of the application. The data model specification of an application is the sum of the data model specifications of a set of classes. Thirdly, we defined a migration process, which translates the old data model specifications into the new structure. The generation process was mostly automatic. Conflict handling was semi-automatic and uses pair reviews. At the end, we integrated the new procedure for defining data models into the modeling tools of the ObjectLens.

Conclusion

In this paper, we described an approach to replace the huge monolithic data model specification of the ObjectLens by modular data model specifications and generated data models. In connection with a product line strategy, the old monolithic OR-mapping design leads to a high degree of redundancy, which complicates development and maintenance.

The main idea of our solution is to describe the mappings of each class in the class itself using inheritance and generate the whole specification from a list of single class data models. In this way, declarative and generative programming techniques are combined.

After some months of productive use, we can claim that we achieved our goals. The proposed solution works well. We migrated all old data model specifications of all our applications to the new procedure. The integration of different domain modules is simplified. Often, only the method *dataModelClasses* needs to be adapted. The class data model specifications lead to uniform specifications with lower definition conflicts. The creation and maintenance of the small class data definitions is much easier then the old copy&paste approach. Furthermore, the support of inheritance leads to a 'single point of definition' approach and reduces redundancy extremely. Refactoring or extending class hierarchies is much easier now.

On the implementation stage, we decided to reuse as much as possible from the ObjectLens. Therefore, the data model mappings of the classes use the same lens literal encoding like the original specifications. The class *DataModelDefinitionGenerator*, which we used at first for the migration process, was also suitable for the generation of the *primDataModelDefinition* methods by the mapping tool. The initial *primDataModelDefinition* methods were generated from the old existing *dataModelSpecs*.

On the tools stage, the lens modeling tools were extended to support class data models. The extended lens editor provides support for editing lens data models, which are constructed from a set of classes. The extended mapping tool supports the generation of the method *primDataModelDefinition*, which is the central part of the definition of the lens structure type of a class.

Literature

[AbCa 1996] Abadi, M.; Cardelli, L.: *"A Theory of Objects"* Springer. New York. 1996.

[ABW 1998] Alpert, S. R.; Brown, K.; Woolf, B.: *"The Design Patterns Smalltalk Companion"* Addison-Wesley. Reading (Massachusetts). 1998.

[BaKi 2004] Bauer, C.; King, G.: *"Hibernate in Action"* Manning. Greenwich. 2004.

[BrWh 1996] Brown, K.; Whitenack, B.: *"Crossing Chasms for Object-Relational Integration"* in: "*Proceedings of the 3rd Conference on the Pattern Languages of Programs*" 1996.

[Cinc 2003a] Cincom Systems: *"VisualWorks: Version 7.2.1, Database Application Developer's Guide".* Cincom Systems. 2003. www.cincom.com/smalltalk

[Cinc 2003b] Cincom Systems: *"VisualWorks: Version 7.2.1, Application Developer's Guide".* Cincom Systems. 2003. www.cincom.com/smalltalk

[CHC 1990] Cook, W.; Hill, W.; Canning, P.: *"Inheritance Is Not Subtyping"* in: *"POPL 1990"* pp. 125-135.

[CoMa 1984] Copeland, G.; Maier, D.: *"Making Smalltalk a Database System"* in: *"SIGMOD Record"* Volume 14. Issue 2. 1984. pp. 316-325.

[Date 1995] Date, C. J.: *"An Introduction to Database Systems"* Volume 1. Addison-Wesley. Reading (Massachusetts). 6. Edition. 1995.

[ElNa 1989] Elmasari, R.; Navathe, S.: *"Fundamentals of Database Systems"* Cummings Publishing. Redwood City. 1989.

[Gems 1996] GemsStone Systems: *"GemStone Documentation: Version 5.0"* GemStone Systems, Inc. Juli 1996.

[GHJ+ 1998] Gamma, E.; Helm, R.; Johnson, R. E.; Vlissides, J.: *"Design Patterns CD: Elements of Reusable Object-Oriented Software"* Addison-Wesley. 1998.

[Howa 1995] Howard, T.: *"The Smalltalk Developer's Guide to VisualWorks"* SIGS. New York. 1995.

[IBL 1998] IBL Ingenieurbüro Letters GmbH: "*Polar(R) : Ein Werkzeug zur Abbildung objektorientierter Strukturen auf relationale Datenbanken (Produktpräsentation)*" in: "*Tagungsband STJA '98: Smalltalk und Java in Industrie und Ausbildung*" 1998.

[KeCo 1996] Keller, W.; Coldewey, J.: *"A Design Cookbook for Business Information Systems"* sd&m report. 1996.

[Knig 2004] Knight, A.: *"Tutorial Using Glorp"* in: *"Proccedings of Smalltalk Solutions' 2004"* 2004. www.glorp.org

[Meye 1997] Meyer, B.: *"Object-oriented Software Construction"* 2. Edition. Prentice Hall. 1997.

[Micr 1998] MicroDoc GmbH: "*MicroDoc Persistence Frameworks für Smalltalk und Java: (Produktpräsentation)*" in: "*Tagungsband STJA '98: Smalltalk und Java in Industrie und Ausbildung*" 1998.

[Parc 1994] ParcPlace Systems: *"VisualWorks: Version 2.0"*. Cincom Systems. 2003.

[Prass 2002] Prasse, M.: *"Entwicklung und Formalisierung eines objektorientierten Sprachmodells als Grundlage für MEMO-OML"* Fölbach. Koblenz. 2002.

[RJB 1999] Rumbaugh, J.; Jacobson, I.; Booch, G.: *"The Unified Modeling Language Reference Manual"* Addison-Wesley. 1999.

[Roos 2003] Roos, R. M.: *"Java Data Objects"* Addison-Wesley. Boston. 2003.

[TheO 1998] The Object People GmbH: "*TOPLink: Persistenzframework für Smalltalk und Java (Produktpräsentation)*" in: "*Tagungsband STJA '98: Smalltalk und Java in Industrie und Ausbildung*" 1998.

[Wool 1994] Woolf, B.: *"Understanding and Using ValueModels"* Whitepaper. Knowledge Systems Corporation. 1994.