The Metabase: Generating Object Persistency Using Meta Descriptions

Marco D'Ambros *and* Michele Lanza
Faculty of Informatics
University of Lugano, Switzerland

Martin Pinzger
s.e.a.l. - software evolution and architecture lab
University of Zurich, Switzerland

Abstract

Because of its language independent metamodel FAMIX, the Moose environment allows researchers to analyze software systems in a uniform way and to exchange data and results. To support interoperability, different interchange file formats such as XMI, CDIF, and recently MSE, have been proposed and used. The use of text files, to exchange data, has three major issues: (1) the files containing the model of a system should always be parsed before being used in Moose, (2) the files must be parsed entirely, even if only a small part is needed, and (3) compatibility is a concern since the old interchange formats are no longer supported in recent versions of Moose.

We propose an alternative approach for FAMIX data exchange, based on object persistency. FAMIX models are stored in a database in a transparent way and they can be remotely accessed. Model entities are retrieved from the database "on-demand", i.e., only the parts needed are read and not the entire model. The mapping between Smalltalk objects and database rows is automatically generated from a meta description, which does not limit the approach to the FAMIX meta model, but to any *meta model defined by a meta description.*

1 Introduction

The FAMIX meta model [1] has been used for several years by researchers to exchange descriptions of software systems. Using a language independent meta model allows them to analyze software systems written in different languages, to apply several tools, and to compare results. In the context of the Moose framework, different file interchange formats have been used to exchange FAMIX models. The first two, *i.e.,* CDIF and XMI [3], are not supported any more in recent versions of MOOSE. The last, and currently used, file format is MSE[1].

We identified three major issues in using text files for exchanging models:

1. *Parsing*. The text file containing the model has to be parsed to import the model into the Moose environment.

2. *Performance*. It is not possible to import only parts of the model, *i.e.,* parse only parts of the text file: The entire text file has to be parsed. For models of large software systems this can have a severe impact on performance.

3. *Compatibility*. Recent versions of Moose do not support old interchange file formats anymore. This has a strong impact on tools based on old versions of Moose.

We propose an approach to exchange models, based on object persistency instead of text files. The technique reads/writes objects from/to a database in a transparent way, avoiding import/export operations. The objects are retrieved "on demand," without the need of reading the entire model.

The persistency mechanism, *i.e.,* the mapping between the database and the actual objects, is automatically generated from a meta model description written in Meta.[2] Therefore, the approach is not limited to FAMIX, but can be used for any meta model described by Meta.

The approach is implemented in Smalltalk, in the context of Moose, using GLORP [2] for the object persistency. The concepts presented in this paper are strictly related to this language and environments, but they can be generalized to other languages and environments, *e.g.,* Java and Hibernate.[3]

Structure of the Paper. In Section 2 we introduce the object persistency technique based on GLORP. We then present our Metabase approach in Section 3 and we provide an example in Section 4. We conclude in Section 5 by summarizing our contributions and discussing future work.

[1] See http://smallwiki.unibe.ch/moose/mseformat/

[2] http://smallwiki.unibe.ch/moose/tools/meta/

[3] http://www.hibernate.org/

2 Object Persistency

The Metabase relies on GLORP[4] for object persistency. GLORP is a simple but powerful object-relational mapping layer for Smalltalk. It allows us to define the mapping between Smalltalk objects and table and rows in a relational database (DB). Once this mapping is defined, objects can be read from and written to the DB in a completely transparent way, without having to write any SQL statement.

Example. We want to define the mapping for simplified versions of FAMIXClass and FAMIXMethod. FAMIXClass has a name, belongs to a package and has a collection of methods, while FAMIXMethod just has a name. We need to create a new class, *i.e.,* `FamixDescriptorSystem`, inheriting from `Glorp.DescriptorSystem`. In this class we add methods to define the structure of the database table corresponding to the FAMIX class and method and to define the mapping between the tables and the classes. This is shown in the following code snippets.

```
tableForFAMIXClass: aTable
 aTable createFieldNamed: 'Id' type: platform serial.
 aTable createFieldNamed: 'Name'
   type: (platform varChar: 50).
 aTable createFieldNamed: 'PackagedIn'
   type: (platform integer).

tableForFAMIXMethod: aTable
 aTable createFieldNamed: 'Id' type: platform serial.
 aTable createFieldNamed: 'Name'
   type: (platform varChar: 50).
 aTable createFieldNamed: 'BelongsTo'
   type: (platform integer).

descriptorForFAMIXClass: aDescriptor
 | t |
 t := self tableNamed: 'Class'.
 tMethod := self tableNamed: 'Method'.
 tAttribute := self tableNamed: 'Attribute'.
 tPackage := self tableNamed: 'Package'.
 aDescriptor table: t.
 aDescriptor addMapping:
   (DirectMapping from: #dbId to: (t fieldNamed: 'Id')).
 aDescriptor addMapping:
   (DirectMapping from: #name to: (t fieldNamed: 'Name')).
 (aDescriptor newMapping: OneToOneMapping)
   attributeName: #packagedIn;
   referenceClass: FAMIXPackage;
   mappingCriteria: (Join from: (t fieldNamed: 'PackagedIn')
     to: (tPackage fieldNamed: 'Id')).
 (aDescriptor newMapping: OneToManyMapping)
   attributeName: #methods;
   referenceClass: FAMIXMethod;
   join: (Join from: (t fieldNamed: 'Id')
     to: (tMethod fieldNamed: 'BelongsTo')).
 ^aDescriptor

descriptorForFAMIXMethod: aDescriptor
 | t |
 t := self tableNamed: 'Method'.
 aDescriptor table: t.
 aDescriptor addMapping:
   (DirectMapping from: #dbId to: (t fieldNamed: 'Id')).
 aDescriptor addMapping:
   (DirectMapping from: #name to: (t fieldNamed: 'Name')).
```

In the code snippet we see three kinds of mapping: Direct, one-to-one and one-to-many.

Direct Mapping. It is used to express simple relationships between instance variables and table columns. It is used when the "type"[5] of the instance variable is directly supported by the DB, for example for Integer, Text, Varchar, Timestamp, *etc.*

One-to-one Mapping. It is used to express the relationship between FAMIXClass and FAMIXPackage. This mapping, declared in the FAMIXClass descriptor, defines the following properties:

- The attribute name: The name of the instance variable getter.
- The reference class: Specifies the class of the objects, and therefore the corresponding table in the DB. In the considered case the class is FAMIXPackage, which corresponds to the Package table (not shown for brevity).
- The join expression: Defines which columns of the two tables are linked. This information is used by GLORP to create the appropriate SQL join query to fetch the data from the DB and create the objects.

One-to-many Mapping. It expresses that a FAMIXClass can have several FAMIXMethods. The structure of the mapping is the same as the one-to-one mapping, with two differences. First, the attribute name refers to a collection of objects instead of a single object. All these objects have to be instances of the class "referenceClass" (FAMIXMethod). Second, the data will be written in the table corresponding to the reference class (FAMIXMethod), instead of the current class (FAMIXClass). This is because each row in the method table refers to a row in the class table (the container class), while each row in the class table can refer to multiple rows in the method table.

A last type of mapping, not used in the code snippet, is *many-to-many*. It expresses the most generic relationship by means of a link table. If two classes have this kind of relationship, the relationship itself is stored in a separated link table in the DB.

What about Inheritance? We want to add to our simplified FAMIX model a superclass of FAMIXClass, namely FAMIXAbstractNamedEntity, which has a name as instance variable. When adding this superclass, we also remove the *name* instance variable from the FAMIXClass class, since it is inherited from FAMIXAbstractNamedEntity. GLORP

[4]Generic Lightweight Object-Relational Persistence. For details refer to [2] and to the GLORP web site http://www.glorp.org/

[5]To use GLORP we have to assume that an instance variable is always of the same class, called type.

provides two techniques to manage inheritance: Filtered and Horizontal. In the filtered inheritance all of the classes are represented in a single table, with a discriminator field for which subclass they are. The table has the union of all possible fields for all classes. In the horizontal inheritance each concrete class is represented in its own table. Each table will duplicate the fields that are in common between the concrete classes. Figure 1 shows the two approaches for our examples.

AbstractNamedEntity	
Id	Name
1	EntityA

Class		
Id	Name	PackagedIn
1	ClassA	PackageA

(a) Horizontal Inheritance

AbstractNamedEntityAndClass			
Id	Name	PackagedIn	Class
1	EntityA	-	FAMIXAbtractNamedEntity
2	ClassA	PackageA	FAMIXClass

(b) Filtered Inheritance

Figure 1. Types of inheritance in GLORP

In horizontal inheritance (Figure 1(a)) the name column is duplicated in both tables, in filtered inheritance (Figure 1(b)) the PackagedIn value is nil for the AbstractNamedEntity EntityA and there is the "Class" identifier column.

Reading & Writing. Once the mapping between the tables and the objects is defined, *i.e.,* the descriptor class is completed, reading and writing objects is straightforward. The following code snippet reads all the FAMIXClass objects from a DB, modifies them and stores them back in the DB.

```
famixClasses := session readManyOf: FAMIXClass.
"the famixClasses objects are modified"
session registerAll: famixClasses.
```

"session" is an object storing the connection with the DB. It is also possible to retrieve only the objects satisfying a given block with:

```
famixClasses := session readManyOf: FAMIXClass
  where: [:each | each isAbstract].
```

When a FAMIXClass is read from the DB, all the classes which have a relationship with it (in our example FAMIXMethod and FAMIXPackage) are retrieved on demand in a transparent way. This means that the message "`readManyOf:`" sent to the session object retrieves only FAMIXClasses, not FAMIXMethods and FAMIXPackages. If we send the getter message "`methods`" or "`packagedIn`" to a FAMIXClass object, the collection of FAMIXMethod objects or the FAMIXPackage object are dynamically read from the DB.

3 The Metabase

The Metabase takes as input a meta-model described in Meta and outputs a GLORP class descriptor, which defines the mapping between the object instances of the meta model, *i.e.,* the model, and the database. Figure 2 shows how the Metabase works.

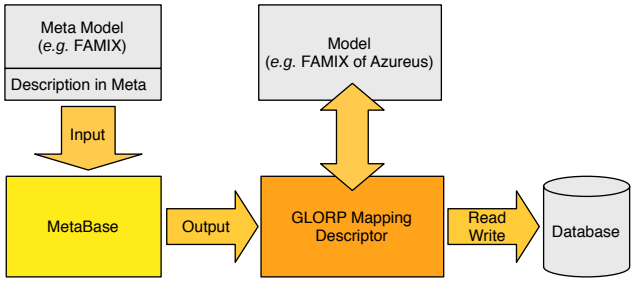

Figure 2. Using the Metabase

Using the Metabase is straightforward. Suppose we have a meta-model described by Meta, *i.e.,* some classes with instance variables described by "meta" methods on the class side. To create the GLORP class descriptor with the Metabase, we can use the following code snippet:

```
classes := OrderedCollection with: ClassA with: ClassB ...
^ClassDBDescriptorGenerator uniqueInstance
  createClassDescriptorForClasses: classes
  named: 'Descriptor' in: aPackage.
```

This code generates the descriptor class named "Descriptor," located in the "aPackage" package. This descriptor can be used to define the mapping between the meta model and the database. To get a connection with the database, which respects the mapping we use the code:

```
db := MetaDB.MetaDBBridge uniqueInstance.
db descriptorClass: Descriptor.
db login: ((Login new)
  username: 'user'; password: 'pass';
  connectString: 'databaseServerLocation';
  database: PostgreSQLPlatform new; yourself).
```

Once the database connection is created, we can create the tables on the database (if the database is empty) with:

```
db createTables
```

and read and write objects of the model with:

```
someClasses := db session readManyOf: ClassA.
someClasses addAll: (db session readManyOf: ClassB).
"someClasses are modified"
db session registerAll: someClasses.
```

Summary. The Metabase generates GLORP descriptors in a fully automatic way. It manages one-to-one, one-to-many and many-to-many relationships among objects. It manages inheritance among meta model classes using filtered inheritance.

4 Example

We present a simple example[6] which shows how the Metabase supports inheritance and all types of relationships (direct, one-to-one, one-to-many, many-to-many).

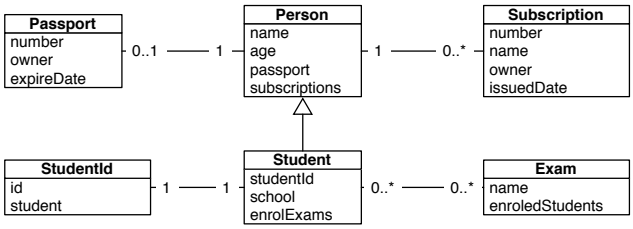

Figure 3. A simplified UML class diagram of the example meta model

Figure 3 shows the UML diagram of the example model. The code snippet below shows the class methods used to describe the meta model in Meta. Due to lack of space we show only the methods for the Person, Subscription, and Passport classes.

```
Person>>metamodelAge
  ^(EMOF.Property name: #age type: Number)
Person>>metamodelName
  ^(EMOF.Property name: #name type: String)
Person>>metamodelPassport
  ^(EMOF.Property name: #passport
    opposite: #owner type: Passport)
Person>>metamodelSubscription
  ^(EMOF.Property name: #subscription opposite: #owner
      type: Subscription multiplicity: #many)
    isDerived: true; yourself.

Subscription>>metamodelIssuedDate
  ^(EMOF.Property name: #issuedDate type: Date)
Subscription>>metamodelName
  ^(EMOF.Property name: #name type: String)
Subscription>>metamodelNumber
  ^(EMOF.Property name: #number type: Number)
Subscription>>metamodelOwner
  ^(EMOF.Property name: #owner
    opposite: #subscription type: Person)

Passport>>metamodelExpireDate
  ^(EMOF.Property name: #expireDate type: Date)
Passport>>metamodelNumber
  ^(EMOF.Property name: #number type: Number)
Passport>>metamodelOwner
  ^(EMOF.Property name: #owner
      opposite: #passport type: Person)
    isDerived: true; yourself.
```

Once the meta description is defined as shown in the code snippet, we can generate the GLORP descriptor, read and write objects instances of the meta model from and to the database as described in the previous Section. Figure 4 shows a screenshot of some database tables automatically generated and populated.

[6]The model of the example can be found in the package "MetaDBTest::ExampleDBs", while the generation of the descriptor is on the class side of the class ClassDBDescriptorGenerator.

Person

dbid	school	studentid	passport	age	name	classtype
1	UZH	2	4	22	Pierazzo	Student
2	USI	3	5	20	Pierone	Student
3	*NULL*	*NULL*	1	27	Marco	Person
4	*NULL*	*NULL*	6	28	Il Puzzone	Person
5	*NULL*	*NULL*	2	31	Jonny Bravo	Person
6	Politecnico	1	3	18	Pierino	Student

PersonExamLink

dbid	personid	examid
1	2	2
2	1	3
3	6	4
4	2	4
5	6	1
6	1	1
7	6	2

StudentId

dbid	id	student
1	1	6
2	3	1
3	2	2

Exam

dbid	name
1	Calculus
2	Algebra
3	Greek
4	Physics

Figure 4. A screenshot of the generated database

5 Conclusions

In this paper we have presented a novel approach to support interoperability in reverse engineering, based on object persistency. Instead of using text files to exchange software system models, we propose the use of the *Metabase*. The Metabase takes a meta model description and automatically generates the object persistency descriptor, *i.e.,* the mapping between the objects (instances of the meta model) and the generated database. We implemented the Metabase on top of the Moose reengineering environment, relying on Meta for the meta model description part and on GLORP for the object persistency part.

Future Work. We have used and tested the Metabase with small and relatively simple meta models (but which already include inheritance and all the types of relationships). We plan to test the Metabase with larger and more complex meta models. We also plan to improve the performance of the Metabase, especially with respect to GLORP proxies and cursors, the bottleneck of the current implementation.

References

[1] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[2] A. Knight. Glorp: generic lightweight object-relational persistence. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 173–174, New York, NY, USA, 2000. ACM Press.

[3] S. Tichelaar, S. Ducasse, and S. Demeyer. FAMIX: Exchange experiences with CDIF and XMI. In *Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000)*, June 2000.