softEnvironment
Umwelt- & sozialverträgliche Software

Technical Manual

Using JOMM

Java Object Model Mapping

Version 1.1.0 by Peter Hirzel

soft **Environment**
Umwelt- & sozialverträgliche Software

Peter Hirzel info@softEnvironment.ch
www.softEnvironment.ch

Document Info

Filename:	*Using_JOMM_V1_1_0.doc*
Template:	*softEnvironment_Document.dot*
Responsible:	*Peter Hirzel*
Classification:	None

Edition History

Date	Version	State	Company	Author	Remarks
15.04.2005	1.0.0	Released	*soft*Environment	Peter Hirzel	Initial
02.10.2005	1.1.0	Released	*soft*Environment	Peter Hirzel	Enumerations & Codes, Referential Integrity, Case 1:1

Contents

Preface

JOMM is the result of an iterative process to design and implement a pragmatic Object-Relational Mapping Framework. After first experiences with TOPLink about a decade ago and an own first implementation in Smalltalk, a complete redesign followed in Java. Later on real project experiences with the concepts and specifications of ODMG, EJB and JDO left their fingerprints in this project as well. Finally the partnership with *Eisenhut Informatik AG* specifically during the project UML/INTERLIS-Editor (see http://www.umleditor.org) brought further insights in point of MDA approaches.
Therefore JOMM is not just another Object/Relational Mapper, but its more a serious trial to enhance the whole process from designing a logical data model in UML and generating different specific physical Models out of it with the UML/INTERLIS-Editor for any other Targets.

This document describes the necessary dependencies and usage of JOMM.

JOMM is an *Open source* Project written entirely in Java.

1 Abstract

JOMM is about mapping between different worlds of models such as

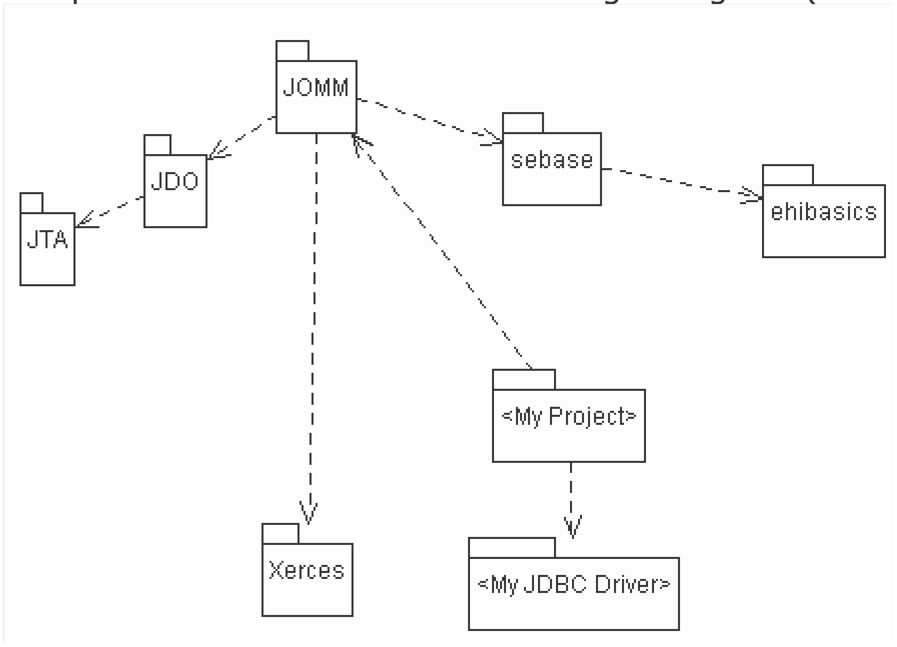
- Logical UML Model (for e.g. Class-diagram)
- Java Model Classes
- SQL relational schema for any Database Server Product (for e.g. MySQL, PostgreSQL, ORACLE, etc)
- XML instance

JOMM is useful in any n-Tier Architecture where one or more specific processes (for e.g. the GUI-Client) are written in Java. Any persistency behaviour from one layer to another must be transformed upon transformation rules. Usually any different Target-System (for e.g. a relational Database Management System) will deal with more or less the same data of a certain problem domain.

Any Java model class with persistency character will have its own descriptor in JOMM, which documents properties, relationships and inheritance in a transformable manner. Thereby some definitions for technical aspects are introduced and assumed with JOMM.

2 Installation

This chapter describes the necessary Software Components to run and compile JOMM as shown in the Package-Diagram (according to UML).



2.1 Java Environment

Java Software Development Kit:

- Based on J2SDK V1.4.* (see http://java.sun.com).
- Based on native JDBC.

2.2 Necessary base libraries

The following packages (jar-libraries) must be included in any JOMM-project separately.

2.2.1 seBase

Base library from *soft*Environment -> sebase.jar

- http://sourceforge.net/projects/umleditor

2.2.1.1 ehibasics

seBase is based on the library by Eisenhut Informatik AG -> ehibasics.jar

- http://sourceforge.net/projects/umleditor

2.2.2 JDO

Java Data Objects (JDO) -> **jdo.jar** (tested with Version 1.0.1)

- http://www.sun.com/software/communitysource/jdo/download.xml

2.2.3 JTA

Java Transaction API (JTA) -> **jta.jar** (tested with Version 1.0.1B)

- http://java.sun.com/products/jta/

2.3 Optional base libraries

The following libraries will be necessary only if a specific Target Transformation is expected.

2.3.1 XML-Mapping

If JOMM shall be used for XML-Mappings, the Apache Xerces Project is used for parsing XML-Files. If this library is not included with JOMM compile errors will happen in package ch.softenvironment.jdo.xml which might be ignored as long no XML-Mappings are needed.

Apache Xerces for Java -> xerces.jar (tested with Version 1.4.4)

- http://xml.apache.org/xerces2-j/

2.3.2 Relational Database Management Systems

The following relational DBMS are implemented with JOMM (based on Standard SQL99).

Any JDBC (or ODBC) Driver must be installed for a desired target DBMS-Product. In the following some tested DBMS are listed, other DBMS might work as well or at least extended with minimal effort by creating new implementations such as ch.softenvironment.jda.<myDBMS>.

2.3.2.1 MySQL

MySQL V4.1.10 JDBC-Driver -> mysql-connector-java-3.2.0-alpha-bin.jar (tested)

- http://www.mysql.com

2.3.2.2 PostgreSQL

PostgreSQL V8.0 JDBC-Driver -> postgresql-8.0.309.jdbc2.jar (tested)

- http://www.postgresql.org/

2.3.2.3 ORACLE

ORACLE 9i JDBC-Driver -> ojdbc14.jar (tested)

- http://www.oracle.com

2.3.2.4 HSQLDB

HSQLDB V1.8.0 JDBC-Driver -> hsqldb.jar (tested)

- http://www.hsqldb.org

2.3.2.5 MS Access

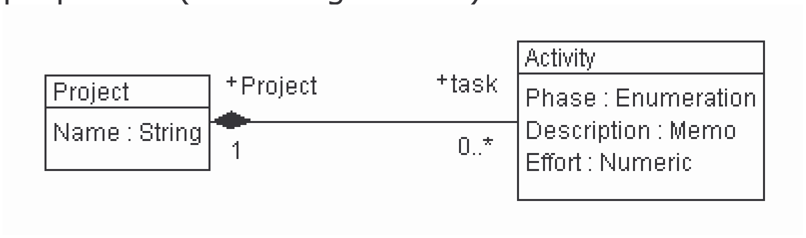
MS Access 2002 -> uses JDBC/ODBC-Bridge (no library must be included specifically)

3 Technical definitions

In relation to MDA (Model Driven Architecture).

3.1 Platform Independent Model (PIM)

For e.g. a two related, persistent classes with problem domain specific properties (according to UML):



No technical fields are to be defined in a PIM.

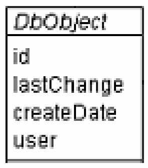
3.2 Platform Specific Model (PSM)

Any Target Platform dependent Model should be generated out of the given PIM (for e.g. with http://www.umleditor.org).

According to the definitions of the target-system some technical attributes may or rather should be added to the PSM (as a transformation of PIM). Dependent of the Target-System's capabilities such technical extensions must be defined from case to case separately. For JOMM these technical extensions are described in this chapter.

3.2.1 Java

Reserved Bean-Properties:



```
java.lang.Long fieldId;
setId();
getId();

java.util.Date fieldLastChange;
setLastChange();
getLastChange();

java.util.Date fieldCreateDate;
setCreateDate();
getCreateDate();

java.lang.String fieldUserId;
setUserId();
getUserId();
```

3.2.2 XML

Because the UML/INTERLIS-Editor (http://www.umleditor.org) already offers an XSD-Generator, JOMM is focused on the principles defined by INTERLIS (see http://www.interlis.ch).
This will make it very easy to generate an XSD out of a PIM and JOMM will map the specific Java-Objects to an XML-Instance which will validate with the very same XSD.

Reserved Attributes:

```
<ch.softenvironment.MyModel TID="x1000003">
…
```

3.2.3 SQL

3.2.3.1 Types

Some databases allow the definition of additional own Domain Types. If possible the following types might be assumed:

Domain Name	Java "Type" (for DbObject)	SQL Type [MySQL]	SQL Type [ORACLE]	SQL Type [MS Access]	Values	Remarks
DbBoolean	java.lang.Boolean	CHAR(1)	CHAR(1)	Text(1)	'T'(rue) or 'F'(alse)	BIT(n) is not supported by all DBMS
DbDate	java.util.Date	DATE			Or DATETIME if DATE is not available	
DbId	java.lang.Long	LONG	NUMERIC(9)	LONG INTEGER	>= 0	
DbTimestamp	java.util.Date	TIMESTAMP (refreshes field at update automatically; once per table)	TIMESTAMP			Used for T_LastChange
DbDateTime	java.util.Date	DATETIME	TIMESTAMP			Used for T_CreateDate
DbUser	String	VARCHAR(30)	VARCHAR2(30)			UserId of a current Session between Client and Server.
DbTable	String	VARCHAR(64) Limited in MySQL to 64 Characters	VARCHAR2(30)			Use the minimal length for Migration-Reasons => **30**

3.2.3.2 Technical Fields

Reserved Attributes in a Model-Schema:

Technical Field	**Description**	**SQL Type**	**Example**
T_Id	UNIQUE Object-Id, by means SQL PRIMARY KEY	DbId	
T_Id_<RoleName>	ForeignKey's are defined by a Prefix + the Rolename of the model.	Same Type as for T_Id	T_Id_Task
T_CreateDate	Date when record was INSERTed the very first time	DbDateTime	
T_LastChange	Date when record was UPDATEd the last time	DbTimestamp	
T_User	Session-User changing the record	DbUser	
T_Type	Inherited Classes. The Root-Class must have this additional field to keep the TableName of the Concrete Subclass Type.	DbTable	
T_Attribute	The Attribute-Name of any Table.	DbTable	
T_Sequence	Attribute to define UNIQUE ordering once per Table. The Ordering is independent of Model Aggregation.	DbID	CREATE TABLE Weekdays (T_ID.., T_Sequence.., T_ID_Name...)
T_Seq_<RoleName>	For Ordered 1:n Relationships		If an Object aggregates many Companies by Role "Company" then they will be ordered according to T_Seq_Company for this very Relationship.
T_Ix_<RoleName>	CREATE INDEX T_IX_MyIndexName…	<none>	
T_Ux_<Constraint>	CREATE UNIQUE INDEX T_UX_MyUniqueConstraint…		

3.2.3.3 Technical Tables

Technical Table Name	**Description**	**MS Access MySQL**	**ORACLE PostgreSQL**	
T_Key_*	Management Table of next free Unique ID for a given Class-type.	T_Key_Object	T_Key_<Sequence>	See chapter 3.2.3.3.1
T_Map_*	Mapping Table's for an Attribute-Mapping with Mulitplicity [0..*]			Having the following Attributes: • T_Id_Owner • T_Type_Owner • T_Attribute_Owner • T_Id_Value
T_NLS	For Multi-language NLS Strings			See chapter 3.2.3.3.2
T_Translation				

3.2.3.3.1 Key-Table

JOMM supports two kinds of obtaining unique Keys in a Multi-User environment.

1) Sequence mechanism (for e.g. ORACLE, PostgreSQL, HSQLDB)
2) Own utility mechanism via a dedicated Key-Table (for e.g. MS Access, MySQL)

3.2.3.3.1.1 With SEQUENCE

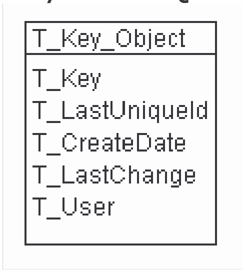
If your DBMS supports SEQUENCE, create a Sequence for any Table having insertables.

T_Key_<TableName> for appropriate **SEQUENCE** Tables for each DbObject which is insertable.

3.2.3.3.1.2 Without SEQUENCE

If your DBMS does not support SEQUENCE, the 2nd variant will assume a Table named **T_Key_Object** and add each DbObject's Table as **T_Key**. The next free UNIQUE ID for a Key-Object will be therefore **T_Id_Last** + 1:

Physical SQL-Schema:

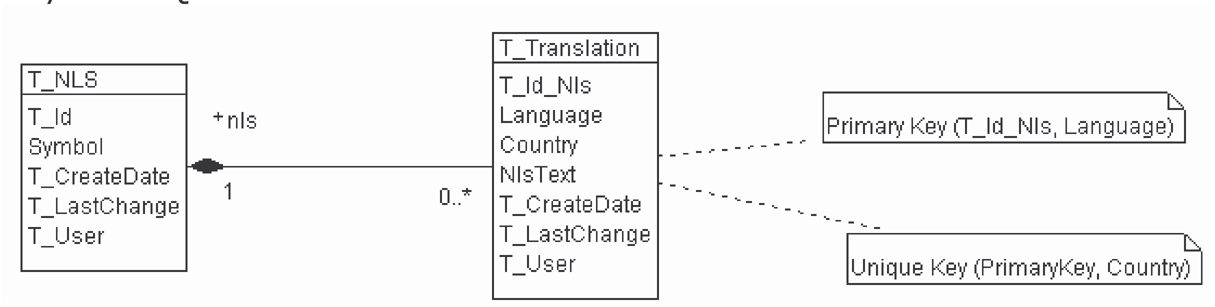


Where T_Key contains the name of any Table of the PSM and T_LastUniqueId is the latest used max. Key given for such an object.

3.2.3.3.2 NLS Support

Support for translatable Strings will be maintained by the following utility shema.

Physical SQL-Schema:



Where Symbol is a general description of what is meant by NLS-Text and T_Translation contains any Translation in another Language (and Country) for a given Symbol.

```
Languages: de; fr; en, etc
Countries : CH ; FR ; GB, etc
```

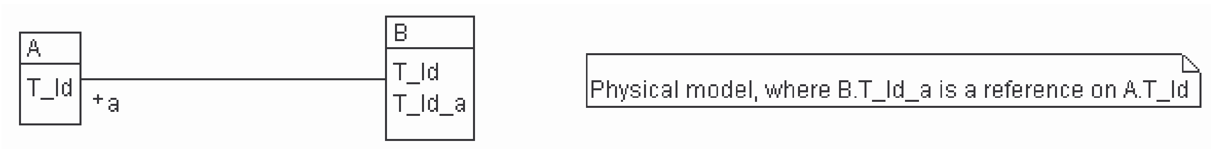
3.2.3.4 Referential Integrity

To make sure constraints are handled consistently within a DBMS or by a Client-Application, any such definitions are assumed to be done by physical Schema!

Though DbDescriptor (see chapter 4.3) makes it possible to define referential types, these definitions are rather for informational use only. Therefore JOMM does not delete any composited objects on the target-server by itself for example, instead a DELETE constraint is assumed on the target side.

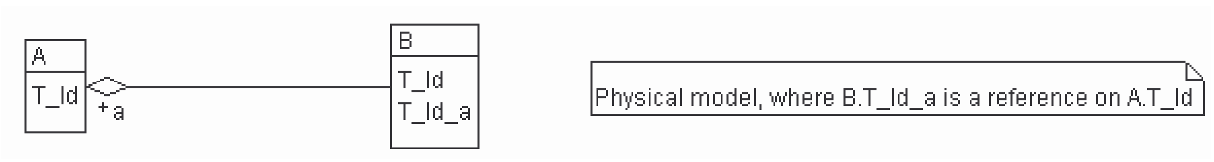
In the following a short understanding of constraints between UML-model and SQL-Schema is outlined:

3.2.3.4.1 Association



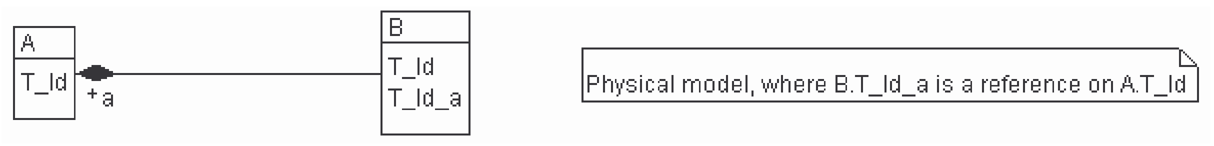
Any Foreign-key role-fields (such as T_Id_a) should be defined by a constraint as: **ON DELETE SET NULL** (from the view of A)

3.2.3.4.2 Aggregation



Any Foreign-key role-fields (such as T_Id_a) should be defined by a constraint as: **ON DELETE RESTRICT** (from the view of A)

3.2.3.4.3 Composition



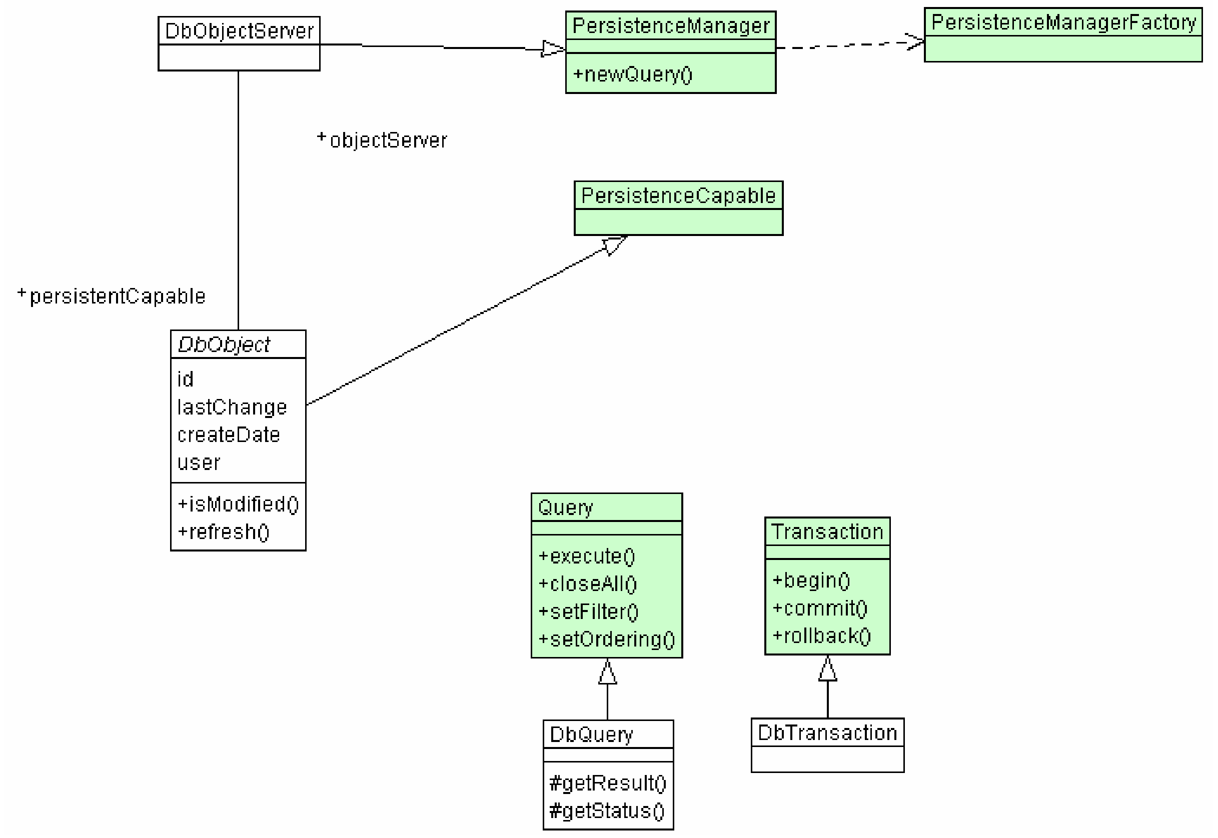
Any Foreign-key role-fields (such as T_Id_a) should be defined by a constraint as: **ON DELETE CASCADE** (from the view of A)

4 Design-Overview of JOMM

This chapter explains the Design and internal implementation of JOMM.

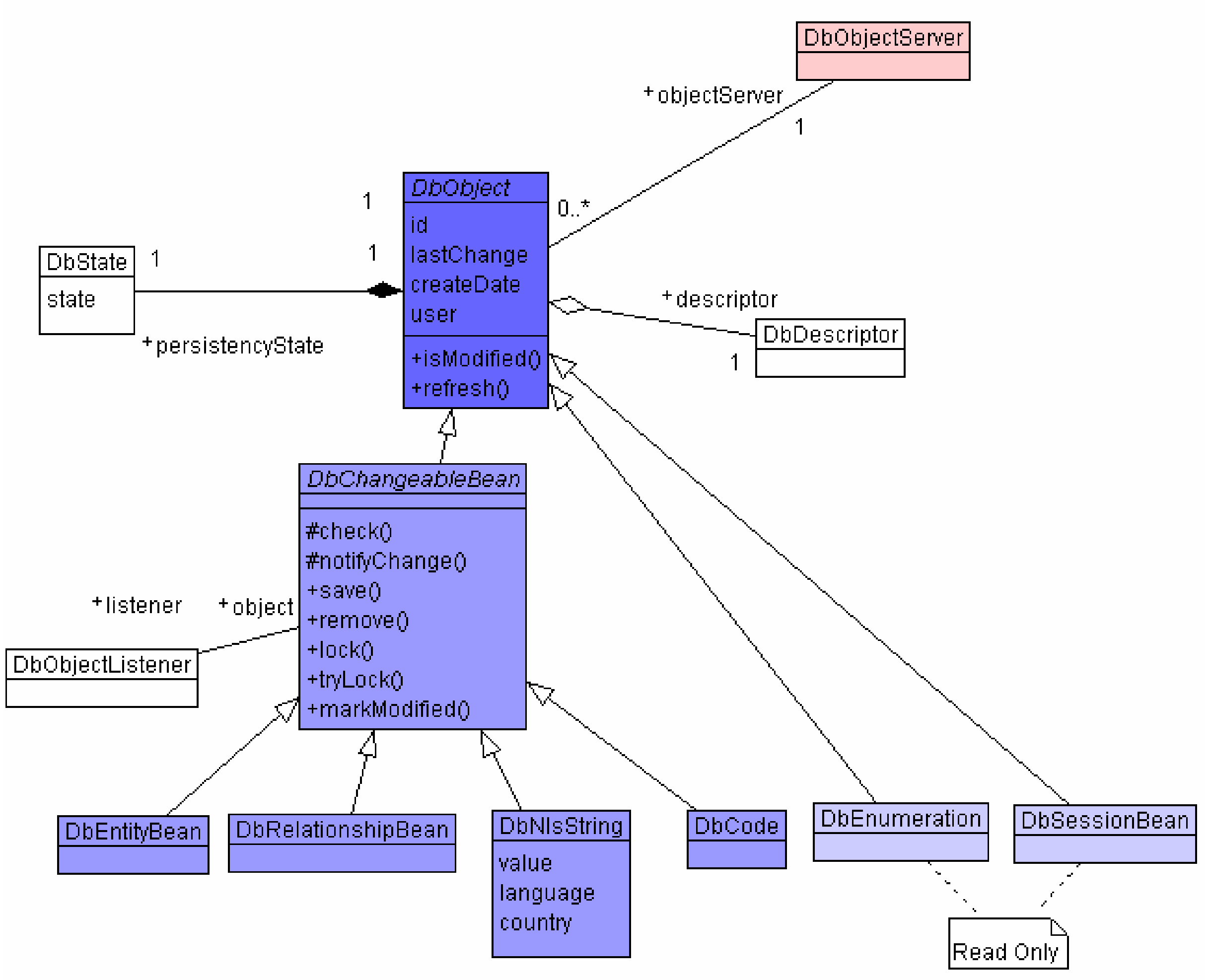
4.1 Implementing JDO-Interfaces

The green classes represent JDO-Interfaces and the white represent JOMM Implementations.



4.2 Persistency Hierarchy

Any persistent Object of the problem domain must inherit from DbObject or one of its predecessors (concepts of EJB and JDO are adapted here).



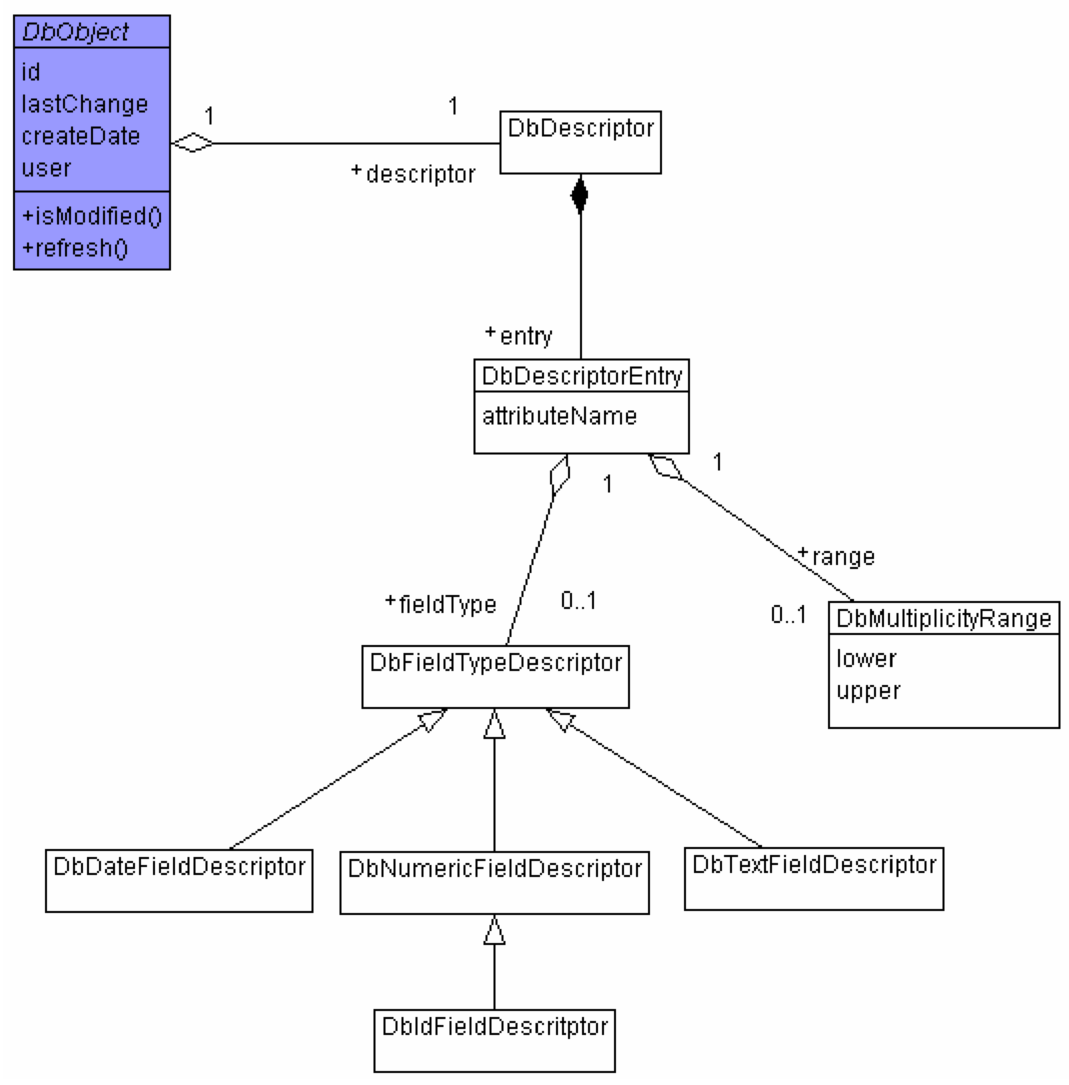
4.3 Descriptor of DbObject

Each persistency class inheriting from DbObject (see package ch.softenvironment.jomm.mvc.model) needs a static DbDescriptor (see package ch.softenvironment.jomm.descriptor) to describe its mappings. Such a description of the mappings is necessary to instantiate or make persistent from Java to other Target-Systems automatically.

The description for the Java-persistency-objects is always the same independent of the used Target-Systems.

Remark:

- Other JDO-Implementations tend to define Model-Descriptions outside of the Java-Source Code for e.g. in a XML-Descriptors.



```
public class MyClass extends DbEntityBean {
     public static DbDescriptor createDescriptor() {
            DbDescriptor descriptor = new DbDescriptor(MyClass.class);
                  …
            return descriptor;
      }
      …
}
```

4.3.1 Technical properties

Typical technical attributes such as Identity (for e.g. #id => T_Id) are mapped by JOMM implicitely, by means such mappings must not be done by any problem domain class.

4.3.2 Problem domain properties

Such properties must be added to the DbDescriptor of each concrete DbObject.

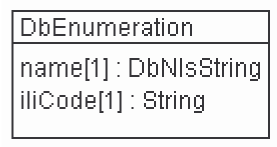
Any properties must be added by DbDescriptor#add*() methods.

4.3.3 Inheritance

Full inheritance is maintained by the DbDescriptor, by means any hierarchical class has its own descriptor mapping of its local attributes and associations.

4.3.4 Enumerations

Enumerations are some kind of Read-Only types in INTERLIS manner (where Codes are non-INTERLIS changeable types), by means they possess an attribute `iliCode` which make each entry unique.
An Enumeration also has a speaking attribute `name` (NLS):



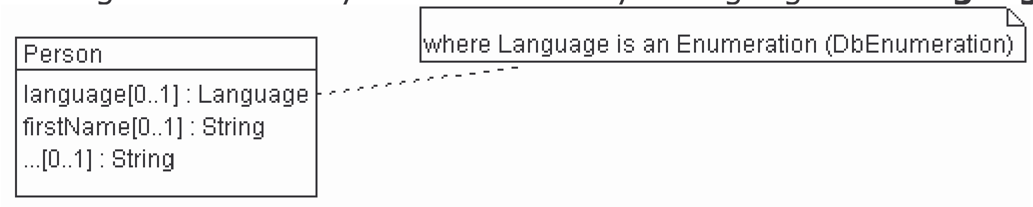
(where length of iliCode-String = 254)

Note:

- In UML/INTERLIS-Editor Enumerations are modelled as "DomainDef's".

4.3.4.1 1:[0..1] Enumeration

For e.g. a Person may be addressed by a language => **language[0..1]**:


Person
language[0..1] : Language
firstName[0..1] : String
...[0..1] : String
where Language is an Enumeration (DbEnumeration)

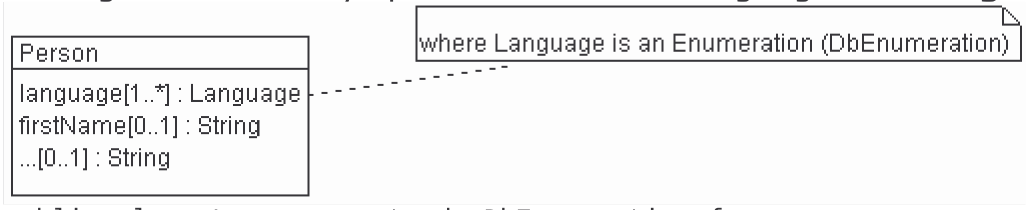
```
public class Language extends DbEnumeration {
  public Language(ch.softenvironment.jomm.DbObjectServer objectServer) {
    super(objectServer);
  }
  public static DbDescriptor createDescriptor(Class dbCode) {
    DbDescriptor descriptor=DbEnumeration.createDescriptor(dbCode);
    return descriptor;
  }
  // ILI-Code constants (UNIQUE)
  public static final String GERMAN="de";
  …
}

public class Person extends DbEntityBean {
  private Language fieldLanguage;

  public static DbDescriptor createDescriptor() {
    DbDescriptor descriptor = new DbDescriptor(Person.class);
    descriptor.addCode("language", "language",
                        new DbMultiplicityRange(0,1));
    descriptor.add("firstName","firstName",
    …
  }
  public Language getLanguage() {
    refresh(false); // read lazy initialized objects
    return fieldLanguage;
  }
  public void setLanguage(Language language){
    Language oldValue=fieldLanguage;
    fieldLanguage=language;
    firePropertyChange("language", oldValue, fieldRole);
  }
  …
}
```

4.3.4.2 1:[0..*] Enumeration

For e.g. a Person may speak more than 1 languages => **language[1..*]**:



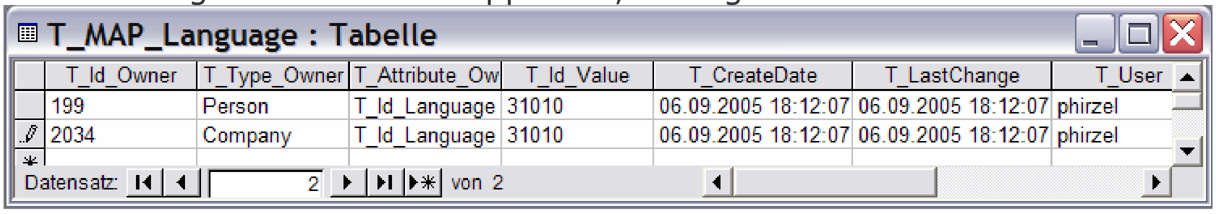
Person
language[1..*] : Language
firstName[0..1] : String
...[0..1] : String
where Language is an Enumeration (DbEnumeration)

```
public class Language extends DbEnumeration {
  (see chapter 4.3.4.1)
}

public class Person extends DbEntityBean {
  private java.util.List fieldLanguage=new java.util.ArrayList();

  public static DbDescriptor createDescriptor() {
    DbDescriptor descriptor = new DbDescriptor(Person.class);
    descriptor.addCode("language", "language",
                  new DbMultiplicityRange(1, DbMultiplicityRange.UNBOUND),
                  Language.class, "T_MAP_Language");
    descriptor.add("firstName","firstName",
    …
  }
  public java.util.List getLanguage() {
    refresh(false); // read lazy initialized objects
    return fieldLanguage;
  }
  public void setLanguage(java.util.List language){
    java.util.List oldValue=fieldLanguage;
    fieldLanguage=language;
    firePropertyChange("language", oldValue, fieldLanguage);
  }
  …
}
```

Persistent Classes which map a [0..*]-Enumeration will need a T_Map_<EnumTable>. Such **n:n generic maps** are to be created for each Enumeration-type referenced in a multiple manner, though several different owners may be kept in this very same table, by means no real relations are modelled in an SQL-schema since enumeration-mappings are considered generic with this approach, for e.g.:



T_MAP_Language : Tabelle

T_Id_Owner	T_Type_Owner	T_Attribute_Ow	T_Id_Value	T_CreateDate	T_LastChange	T_User
199	Person	T_Id_Language	31010	06.09.2005 18:12:07	06.09.2005 18:12:07	phirzel
2034	Company	T_Id_Language	31010	06.09.2005 18:12:07	06.09.2005 18:12:07	phirzel

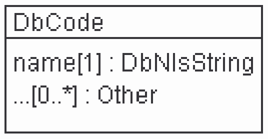
Datensatz: 2 von 2

where:

- T_Id_Owner => id of owning Object mapping the enumeration (Person.T_Id mapping Language)
- T_Type_Owner => will be set to ROOT of inheritance chain of owning object (Person is the root Object here)
- T_Attribute_Owner => Pseudo Foreign-Key (n:n from DbEntityBean to Enumeration)
- T_Id_Value => Id of referenced Enumeration (Language.T_Id)

4.3.5 Codes

A code is a user-changeable type (where an Enumeration is RO) and has a name (NLS) attribute and optional application-specific attributes.



For Descriptor samples see chapter 4.3.4.

4.3.6 Associations

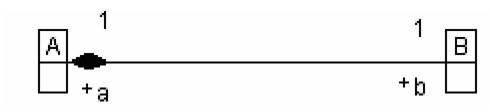
Relationships between classes can be fully described by the DbDescriptor on each side of connected nodes.

An **AssociationEnd** may be of the following type:

- `ASSOCIATION`
- `AGGREGATION`
- `COMPOSITION`

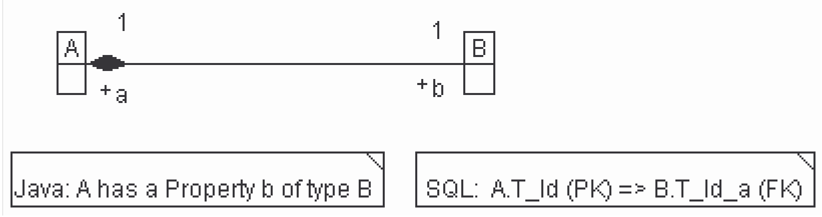
4.3.6.1 Case 1:1

For all (referenced, aggregated or composited) combinations of A:B=[0..1]:[0..1]



4.3.6.1.1 Variant Reference defined in B (preferred)

Where A.T_Id becomes a Foreign Key in B => B.T_Id_a (Role of +a as SQL-Reference-field in B):


A
1
+a
B
1
+b
Java: A has a Property b of type B
SQL: A.T_Id (PK) => B.T_Id_a (FK)

Remark:

- This is the preferred variant because it is **easily changeable into 1:n** if necessary at any time in point of the physical schema.

Usage within JOMM:

```
class A extends DbEntityBean {
      public static DbDescriptor createDescriptor() {
            DbDescriptor descriptor = new DbDescriptor(A.class);
            descriptor.addOneToOne(..);
            return descriptor;
      }
}

class B extends DbEntityBean {
      public static DbDescriptor createDescriptor() {
            DbDescriptor descriptor = new DbDescriptor(B.class);
            descriptor.addOneToOneReference(..);
            return descriptor;
      }
}
```

DbDescriptor Method	Description
`#addOneToOne()`	Use **on side A** where no physical reference of B exists. An instance of B will be mapped in A when A is instantiated.
`#addOneToOneReference()`	Use **on side B** where a physical reference of A exists. An instance of A will be mapped in B when B is instantiated.
`#addOneToOneReferenceId()`	Use **on side B** where a physical reference of A exists. Only Id of A will be mapped in B when B is instantiated for more efficency.

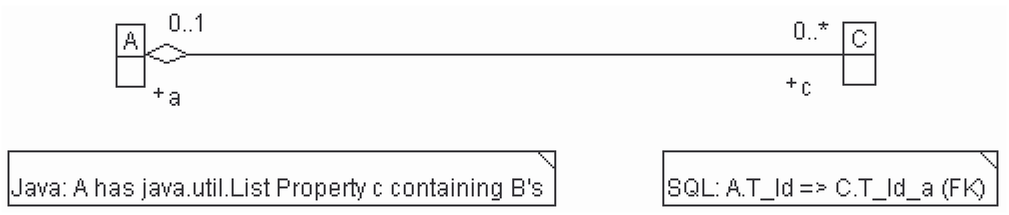
4.3.6.1.2 Variant Reference defined in A (non-preferred)

Where B.T_Id becomes a Foreign Key in A => A.T_Id_b (Role of +b as SQL-Reference-field in A)

DbDescriptor Method	Description
`#addOneToOneId()`	Use **on side B** where no physical reference of A exists.
`#addOneToOneReference()`	Use **on side A** where a physical reference of B exists.
`#addOneToOneReferenceId()`	Use **on side A** where a physical reference of B exists.

4.3.6.2 Case 1:n

- for all combinations of A:B=[0..1]:[0..*], where A.T_Id becomes a Foreign Key in B => B.T_Id_RoleOfA


A
0..1
+a
0..*
C
+c
Java: A has java.util.List Property c containing B's
SQL: A.T_Id => C.T_Id_a (FK)

Usage within JOMM (see chapter ???):

```
class A extends DbEntityBean {
      public static DbDescriptor createDescriptor() {
            DbDescriptor descriptor = new DbDescriptor(A.class);
            descriptor.addOneToMany(..);
            return descriptor;
      }
}

class C extends DbEntityBean {
      public static DbDescriptor createDescriptor() {
            DbDescriptor descriptor = new DbDescriptor(C.class);
            // for the whole Object of A

            descriptor.addManyToOneReference(..);
            // or for the key reference T_Id_a only
            descriptor.addManyToOneReferenceId(..);
            return descriptor;
      }
}
```

4.3.6.3 Case 1:n Ordered

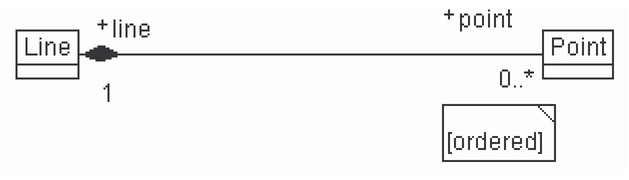


Table Point will have an Attribute allowing to specify the desired ORDER:

- T_Seq_Line

4.3.6.4 Case n:n

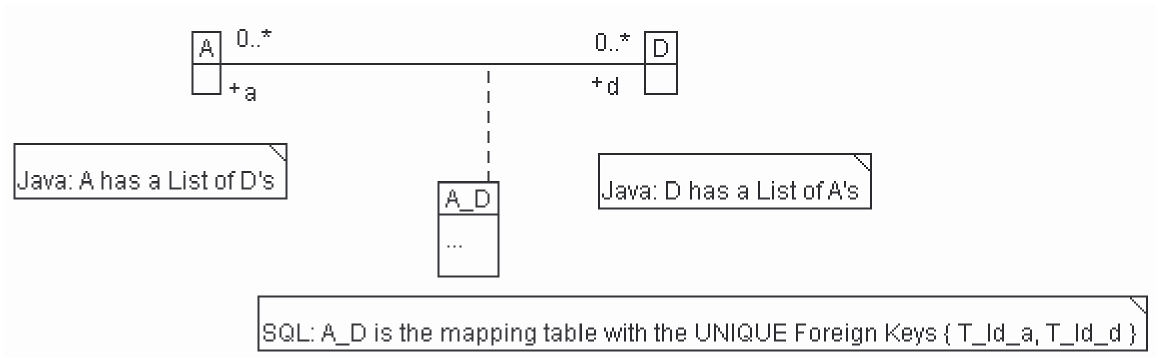
- for all combinations of A:D=[0..*]:[0..*]

Many to Many Relationships become **"Attributed Associations"**, by means A:D=n:n is mapped by an Association-Class/Table A_D.

A:D=n:n via Map C
A.d's => java.util.List owning Object's of type DbRelationshipBean A_D
D.a's => dito

Remark:

- The UML-Editor shows an AssociationClass automatically if additional Attributes are defined on the Association.


A
0..*
+a
0..*
D
+d
Java: A has a List of D's
A_D
...
Java: D has a List of A's
SQL: A_D is the mapping table with the UNIQUE Foreign Keys { T_Id_a, T_Id_d }

The AssociationClass A_D has is its own Object-Identity (by means in SQL: A_D.T_Id (PK)) for e.g. to support Relationship-to-Relationship. Additionally in SQL the combination of Foreign Keys in A_D pointing to the AssociationEnds must be UNIQUE

Usage within JOMM:

```
class A extends DbEntityBean {
      public static DbDescriptor createDescriptor() {
            DbDescriptor descriptor = new DbDescriptor(A.class);
            descriptor. addAssociationAttributed(DbDescriptor.ASSOCIATION
                  “dId”,
                  new DbMultiplicityRange(0, DbMultiplicityRange.UNBOUND),
                  new DbMultiplicityRange(0, DbMultiplicityRange.UNBOUND),
                  A_D.class,
                  "aId");
            return descriptor;
      }
}

class D // analog A

class A_D extends DbRelationshipBean {
      public static DbDescriptor createDescriptor() {
            DbDescriptor descriptor = new DbDescriptor(A_D.class);
            descriptor.addAssociationEnd(A.class, “aId”, “T_Id_a”);
            descriptor.addAssociationEnd(D.class, “dId”, “T_Id_d”);

            // evtl. other attributes

            return descriptor;
      }
}
```

DbDescriptor Method	**Description**
#addAssociationAttributed()	Use **on side A** (resp. D) for a list of D’s (resp. A’s).
#addAssociationEnd()	Use **on A_D** where a physical references of A and D are kept in an association class. Therefore both ends must be added to A_D.

4.3.6.5 Relationship to Relationship

FUTURE USE

4.3.6.6 Inherited Relationship

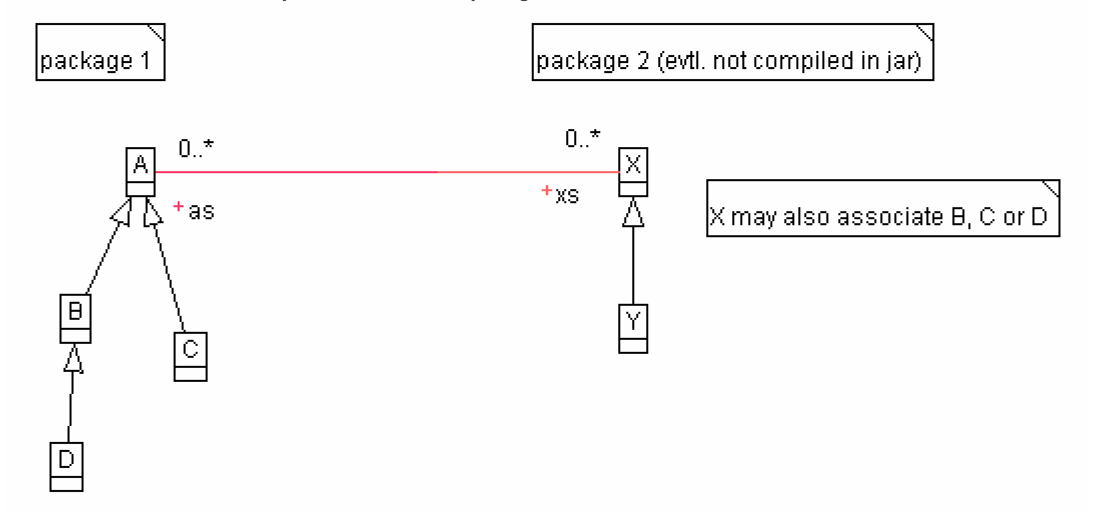
FUTURE USE

4.3.6.7 n-ary

FUTURE USE

4.3.6.8 Associations over "remote" packages

FUTURE USE remote means, packages containing Persistent Classes are not maintained by the same project.


package 1
package 2 (evtl. not compiled in jar)
0..*
0..*
A
X
+as
+xs
X may also associate B, C or D
B
C
D
Y

```
add(.., "package.Class-Name", Attribute);
```

Generic Evaluation via:

```
Object get(Property);
void set(Property, Object);
```

5 Application Development

This chapter describes how to build a Java-Application based on JOMM.

5.1 Using the appropriate Object-Server

Any Java-Process (for e.g. Client-Application) uses a certain DbObjectServer to handle all requests against a Target-System. Therefore an appropriate DbObjectServer implementation should be chosen at connection time in a Java-Application.

```
javax.jdo.PersistenceManagerFactory pmFactory =
      new MySqlObjectServerFactory(); // or any other Implementation
pmFactory.setConnectionURL(url /* + "?user=" + userId + "&password="*/);
pmFactory.setNontransactionalRead(false);// NO autoCommit while reading
pmFactory.setNontransactionalWrite(false);// NO autoCommit while writing
DbObjectServer server =
     DbObjectServer)pmFactory.getPersistenceManager(userId, password);
```

5.2 Establish a Database-Connection

```
DbObjectServer.open(…)
```

Register all Persisent DbObject’s, to bind them for later usage (Descriptor registration):

```
DbObjectRegistry.registerClass(<myapp>.Project.class, "Project");
DbObjectRegistry.registerClass((<myapp>.Activity.class, "Activity");
```

5.3 Dealing with persistency Objects

5.3.1 Instantiating DbObject’s

```
server.createInstance(MyDbObject.class);
```

5.3.2 Changing DbObject’s

```
myDbChangeableBean.save();
```

5.3.3 Removing DbObject’s

Deletion assumes DBMS Referential Integrity (see chapter 3.2.3.4):

- for **inherited DbObject’s the root Object (Table-Entry) will be deleted** by Application only

```
myDbChangeableBean.remove();
```

5.3.4 Enumerations (DbEnumeration)

```
Language german = server.retrieveEnumeration(Language.class, “de”);
```

5.3.5 Code's (DbCode)

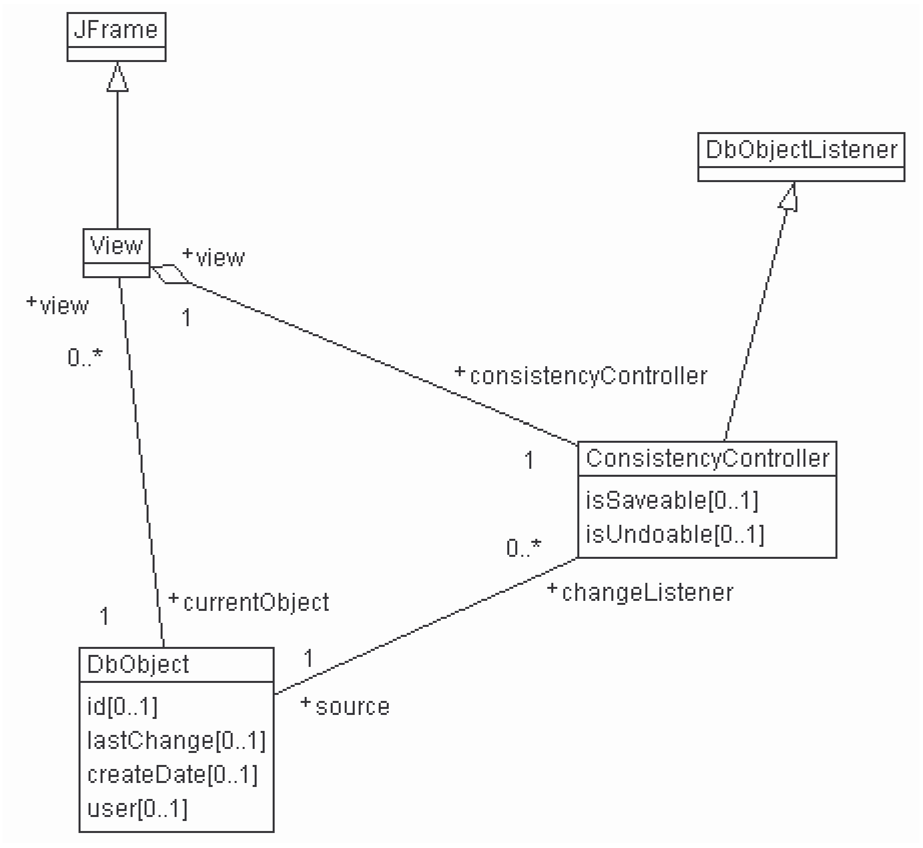
```
java.util.List languages = server.retrieveCodes(Language.class);
```

5.4 Executing a Query

DbQueryBuilder
DbQuery
DbTransaction

5.5 Checking Consistency of persistency Objects

MVC-Pattern



View for e.g. in a GUI to change a Project's properties:

```
class ProjectDetailView() extends javax.swing.JFrame() {
…

public void setCurrentObject(java.lang.Object object) {
      try {
            setObject((Project)object);

            getObject().addPropertyChangeListener(new
            ch.softenvironment.jomm.mvc.controller.ConsistencyController(th
            is));
      } catch(Throwable e) {
            handleException(e);
      }
}
```

```
/**
   * Overwrites.
   */
public void dispose() {
      getObject().removePropertyChangeListener(getConsistencyController());
      super.dispose();
}
```