DB-Main Manual Series

Java Interface for DB-Main

Reference manual

Version 9 - June 2011

The University of Namur - LIBD
REVER s.a.

CONTENTS

CONTENTS I

LIST OF FIGURES III

LIST OF TABLES V

CHAPTER 1
WARNINGS 1

CHAPTER 2
INTRODUCTION 3
2.1 JIDBM 3
2.2 Architecture 3
2.3 JIDBM principles 3
2.3.1 Getting started 3
2.3.1.1 Windows 4
2.3.1.2 Linux 4
2.3.2 Classpath 4
2.3.3 Debugging 4
2.3.4 A simple example 4
2.3.4.1 Loading a project 5
2.3.4.2 Creating a DBMProject 5
2.3.4.3 Multiple project management 5
2.3.4.4 Retrieving schemas from the current project 5

CHAPTER 3
CLASS SPECIFICATIONS 7
3.1 JIDBM repository architecture 7
3.1.1 Naming conventions 7
3.1.1.1 Class naming 8
3.1.1.2 Field naming 8
3.1.1.3 Method naming 8
3.1.2 The project view 10
3.1.2.1 DBMProject 10
3.1.2.2 DBMMetaObject and DBMMetaProperty 10
3.1.2.3 DBMProduct and DBMConnection 12
3.1.2.4 DBMText and DBMTextLine 13
3.1.2.5 DBMSchema 13
3.1.2.6 DBMProductSet 14
3.1.3 The data view 14
3.1.3.1 DBMSchema 14
3.1.3.2 DBMCollection 14
3.1.3.3 DBMDataObject 15
3.1.3.4 DBMEntityRelationshipType 15
3.1.3.5 DBMEntityType, DBMCluster and DBMSubType 15
3.1.3.6 DBMRelationshipType and DBMRole 15
3.1.3.7 DBMAttribute 16
3.1.3.8 DBMSimpleAttribute 16
3.1.3.9 DBMCompoundAttribute 17
3.1.3.10 DBMProcessingUnit 17
3.1.3.11 DBMGroup, DBMConstraint and DBMConstraintMember 17
3.1.3.12 DBMAttributeOwner 18
3.1.3.13 DBMProcessingUnitOwner 18
3.1.4 The process view 19

3.1.4.1 DBMSchema 19
3.1.4.2 DBMDataObject 19
3.1.4.3 DBMProcessingUnit 20
3.1.4.4 DBMState 20
3.1.4.5 DBMElement and DBMElementRelation 20
3.1.4.6 DBMEnvironment 21
3.1.4.7 DBMResource and DBMResourceSubType 21
3.1.4.8 DBMConsumption, DBMProcessingUnitCardinality and DBMResourceCardinality 22
3.1.5 The concrete view 22
3.1.5.1 DBMGenericObject 22
3.1.5.2 DBMConcreteObject 25
3.1.5.3 DBMUserView and DBMUserObject 25
3.1.5.4 DBMNote and DBMNoteRelation 26
3.1.6 The inheritance view 26
3.2 Special classes 26
3.2.1 DBMLibrary 26
3.2.2 DBMConsole 27
3.2.3 DBMClassLoader 27

CHAPTER 4
PROGRAMMING STYLES 29

CHAPTER 5
EXAMPLES 31
5.1 Statistic generator 31
5.1.1 Description 31
5.1.2 Java code 31
5.2 Schema creator 32
5.2.1 Description 32
5.2.2 Java code 32

List of Figures

Figure 3.1 - The project view of the DB-Main repository for JIDBM. 10
Figure 3.2 - The data view of the DB-Main repository for JIDBM. 14
Figure 3.3 - The process view of the DB-MAIN repository for JIDBM. 19
Figure 3.4 - The concrete view of the DB-MAIN repository for JIDBM. 22
Figure 3.5 - The inheritance view of the DB-MAIN repository for JIDBM. 26

LIST OF TABLES

Table 3.1 - The predefined meta-objects in DB-MAIN repository. 11
Table 3.2 - Constants of the fields type and function in the DBMMetaProperty class. 11
Table 3.3 - Constants of the field type in the DBMConnection class. 13
Table 3.4 - Constants of the field typeFile in the DBMText class. 13
Table 3.5 - Constants of the field type in the DBMSchema class. 13
Table 3.6 - Constants of the field setType in the DBMAttribute class. 16
Table 3.7 - Constants of the field type in the DBMSimpleAttribute class. 16
Table 3.8 - Constants of the field type in the DBMProcessingUnit class 17
Table 3.9 - Constants of the field type in the DBMGroup class. 17
Table 3.10 - Constants of the field type in the DBMConstraint class. 18
Table 3.11 - Constants of the field memberRole in the DBMConstraintMember class. 18
Table 3.12 - Constants of the field mode in the DBMProcessingUnit class. 20
Table 3.13 - Constants of the field type in the DBMElement class. 21
Table 3.14 - Constants of the fields type and mode in the DBMEnvironment class. 21
Table 3.15 - Constants of the field objectType in the DBMGenericObject class. 23
Table 3.16 - Constants of the field flag in the DBMGenericObject class. 23
Table 3.17 - Constants of the field type in the DBMUserView class. 25

Chapter 1

Warnings

In this manual, we suppose that JAVA and DB-MAIN concepts are known. The reader not familiar with these concepts is advised to read the following books:

- Horstmann, C. S., Cornell, G., *Core Java 2, Volume-I – Fundamentals (seventh edition)*, Sun Microsystems Press, October 2004.
- *DB-MAIN 9: the modelling tool for your information system – Reference Manual*, DB-MAIN technical documentation, June 2011.
- *DB-MAIN 9: HTML Help*, DB-MAIN technical documentation (F1 in the DB-MAIN CASE tool), June 2011.

Readers interested by more detailed information about the description of the *jidbm* package can also refer to:

- *jidbmdoc: Javadoc of the jidbm package*, DB-MAIN technical documentation, June 2011.

Chapter 2

Introduction

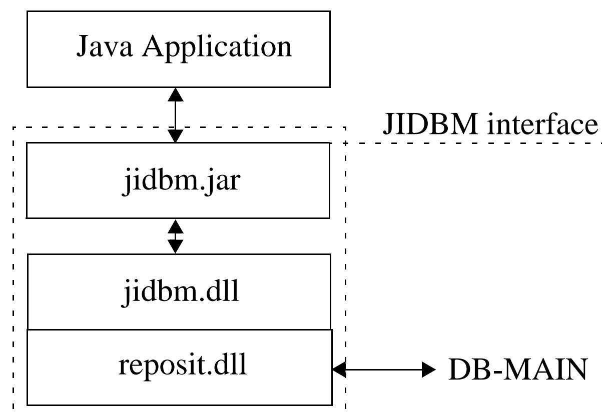
2.1 JIDBM

JIDBM (Java Interface for DB-Main) is a Java API for accessing to the DB-MAIN repository in read and write mode. It consists of a set of classes written in Java programming language. JIDBM provides a Java API for DB-MAIN users and makes possible the writing of applications using a pure Java API.

2.2 Architecture

JIDBM operates with two libraries *jidbm.dll* and *jidbm.jar*:

- *jidbm.dll*: it allows JIDBM that runs within the JAVA Virtual Machine to operate with reposit.dll (the main library of the DB-MAIN repository).
- *jidbm.jar*: the **com.dbmain.jidbm** package containing classes that allow programmers to access to the DB-MAIN repository.



2.3 JIDBM principles

2.3.1 Getting started

The first thing you do is check that you are set up properly. This involves the following steps:

2.3.1.1 Windows

- Java Virtual Machine:
  - If you have the full DB-MAIN installation, DB-MAIN uses the Java Virtual Machine (jvm.dll) in the directory \java\jre\bin\client of the DB-MAIN setup directory.
  - If Java is not present in the DB-MAIN setup directory, you must have Java 1.4 (or later) on your machine. To install Java, simply follow the instructions for downloading the JDK 1.4 (or later). The path to the *jvm.dll* file (for example: C:\Program Files\J2sdk1.4.1_03\Jre\Bin\Client) must be added to the *path* environment variable.
- JIDBM interface: The JIDBM classes are included in the *jidbm.jar* file. This file must be also in the DB-MAIN binary directory (with the *db_main.exe* file). This jar file is provided with DB-MAIN setup file. To develop java plug-ins, it must be interesting to include the *jidbm.jar* file in your *classpath* variable.

2.3.1.2 Linux

- Java Virtual Machine: DB-MAIN uses the Java Virtual Machine (libjvm.so) in the directory /java/jre/lib/i386/client of the DB-MAIN setup directory.
- JIDBM interface: The JIDBM classes are included in the *jidbm.jar* file. This file must be also in the DB-MAIN binary directory (with the *db_main* file). This jar file is provided with DB-MAIN setup file. To develop java plug-ins, it must be interesting to include the *jidbm.jar* file in your *classpath* variable by using something like this in the .bashrc file:
  export CLASSPATH=.:$DB_MAIN_BIN/jidbm.jar:$CLASSPATH

2.3.2 Classpath

By default, each java plug-in developed with the JIDBM interface uses the system variable *classpath* defined on the current machine.

However it is possible to define a local classpath in a file named *.jidbmclasspath* in the plug-in directory. The syntax of this file is the same as the classical classpath (file or directory localization separated by semicolon). In this case, only the local classpath is taken into account to find the libraries used by the java plug-in.

IMPORTANT: Use only absolute paths in *.jidbmclasspath* file and write a single line without carriage return at the end.

2.3.3 Debugging

Remote debugging can be useful for java plug-in development. To achieve that, you must start DB-MAIN in command line mode with two parameters:

```
db_main -pXXX -s[yn]
```

where ***p*** is the transport address for the connection and ***s*** the suspending mode. A development platform (like Eclipse) can listen for a connection at port *XXX* and debug a java plug-in launched in DB-MAIN. if suspending mode is *y*, the target VM will be suspended until the debugger application connects. Otherwise, the debugger stops at the first breakpoint.

2.3.4 A simple example

This chapter goes through a simple example of using JIDBM to access to the DB-MAIN repository. There are two ways to access to the repository:

- Open or create a project in DB-MAIN and run the java program from DB-MAIN.
- Run the java program that loads a project in the repository. In this case, only the dbmfunc.dll, reposit.dll, jidbm.dll and jidbm.jar files are necessary for Windows (libdbmfunc.so, libreposit.so, libjidbm.so and jidbm.jar for Linux).

2.3.4.1 Loading a project

If the java program is launched from DB-MAIN, the first thing you need to do is load (or create) a project you want to use in the DB-MAIN CASE tool.

If the java program is executed independently of DB-MAIN, it must load the project you want to use. Do not forget to delete the current project before the program end in order to free memory and delete all temporary files created during the loading. For example:

```
DBMLibrary lib = new DBMLibrary();
int i = lib.loadLUN(c:\\temp\\test.lun);
if (i > 0) {
  DBMProject pro = new DBMProject();
  if (pro != null) {
    ...
    pro.deleteProject();
  }
}
```

2.3.4.2 Creating a DBMProject

Once the project is loaded, an instance of the object DBMProject can be created in the Java program.

```
DBMProject pro = new DBMProject();
```

2.3.4.3 Multiple project management

If the java program is executed independently of DB-MAIN, it must be possible to load several projects in the DB-MAIN repository. For example:

```
DBMLibrary lib = new DBMLibrary();
int i = lib.loadLUN(c:\\temp\\test.lun);
DBMProject pro1 = new DBMProject();
i = lib.loadLUN(c:\\temp\\test1.lun);
DBMProject pro2 = new DBMProject();
DBMProject curpro = lib.getCurrentProject();
```

Each object of the DB-MAIN repository is identified by its object identifier (local identifier of a project obtained by method *getObjectIdentifier* of class *DBMGenericObject*) and its project identifier (obtained by method *getProjectIdentifier* of class *DBMGenericObject*).

The method *getCurrentProject* of class *DBMLibrary* returns the last project accessed by any class method in the DB-MAIN repository. In the example below, the variable *curpro* is equal to *pro2*.

2.3.4.4 Retrieving schemas from the current project

A DBMProject object represents the project and provides basic methods for accessing repository objects. For instance, it offers several methods for accessing to the main products of a project: *DBMSchema*.

As an illustration[1], the first version of program *Sample* must be executed from DB-MAIN after the loading of a project. Note that the name of the method called by DB-MAIN (to execute a Java plug-in menu File/Execute plug-in…) must be "*runDBM*" (without any parameters). This program accesses all the schemas of a project.

```
class Sample {
    public static void runDBM() throws IOException {
```

1. See chapter 5 for more complete examples.

```
        DBMProject pro = new DBMProject();
        if (pro != null) {
            DBMSchema sch = pro.getFirstProductSchema();
            if (sch != null) {
                // analyze the schema ...
                sch = pro.getNextProductSchema(sch);
            }
        }
    }
}
```

The second version of program *Sample* must be executed in a console. The "*main*" method has a parameter (path and name of a LUN file).

```
class Sample {
    public static void main (String[] args) throws IOException {
        if(args.length != 1){
            System.err.println("usage : Sample <file_name>");
            System.exit(1);
        }
        DBMLibrary lib = new DBMLibrary();
        // load the project in file named args[0] in DB-MAIN repository
        int i = lib.loadLUN(args[0]);
        if (i < 0) {
            System.out.println("The lun file does not exist.");
            System.exit(-1);
        }
        DBMProject pro = new DBMProject();
        if (pro != null) {
            DBMSchema sch = pro.getFirstProductSchema();
            if (sch != null) {
                // analyze the schema ...
                sch = pro.getNextProductSchema(sch);
            }
            // save the project into file named args[0].
            lib.unloadLUN(pro.getProjectIdentifier(),args[0]);
            pro.deleteProject();
        }
    }
}
```

Chapter 3

Class specifications

Section 3.1 describes the JIDBM class architecture and section 3.2 gives some information about special classes of the JIDBM library.

3.1 JIDBM repository architecture

This section presents the DB-Main repository structures for the JIDBM library. This repository is shown as an extended ER[1] schema containing entity types (or classes), relationship types (or rel-types) and attributes (or fields).

The repository is too large to be shown on a single page. For this reason, its definition has been divided into five views according to the ontology that it can model:

- The *project view* (section 3.1.2) models the project window in the DB-MAIN CASE tool.
- The *data view* (section 3.1.3) corresponds to the representation of the ER and UML class diagram schemas.
- The *process view* (section 3.1.4) describes the model for processing schema like UML activity and use case diagrams.
- The *concrete view* (section 3.1.5) shows how graphical, description and note information are stored in the repository.
- The *inheritance view* (section 3.1.6) is an overview of the inheritance mechanism in the repository.

In the following sections, only descriptions of repository objects are given. The reader, interested by further information on interface definition, fields and methods, must refer to the javadoc of the JIDBM package (distributed with DB-MAIN in the Documentation\Manuals\Jidbm directory).

Note that the repository is built to be as general as possible, but DB-MAIN does not always use all its capabilities. Constructs not used by DB-MAIN are reported below. Some of these constructs can be used by a Java program without damage for DB-MAIN, but using other constructs may be hazardous, as mentioned when it is the case.

3.1.1 Naming conventions

In the JIDBM package, the following naming conventions are used.

1. Entity/Relationship.

3.1.1.1 Class naming

The class names are prefixed by DBM and the first letter of any subsequent word are capitalized.

Example: *DBMProductSet* (see figure 3.1 in section 3.1.2).

3.1.1.2 Field naming

The field names start with a lower-case letter and the first letter of any subsequent word is capitalized. The names of fields being used as constants are all upper-case.

Examples (see figure 3.1 in section 3.1.2): field *fileType* of *DBMText* class can get the constant value *FILE_GEN_DIC* (dictionary report file).

3.1.1.3 Method naming

The method names start with a lower-case letter and the first letter of any subsequent word are capitalized. The methods are divided into three categories:

a) Method on fields

The method to get or update class fields are called *getField()* or *setField(new value)*.

Examples (see figure 3.1 in section 3.1.2): *getCreationDate()* and *setCreationDate(...)* of *DBMProject*.

There are three exceptions to this rule:

1. A method to get a boolean field is called *isField()*.
   Example (figure 3.2 in section 3.1.3): *isStable()* of *DBMSimpleAttribute*.
2. Some fields are integers considered arrays of bits where each cell indicates if a constant field is verified or not. In these cases, the method verifying the constant is called *isFieldConstant(constant name)*.
   Example (see figure 3.1 in section 3.1.2): *isFunctionConstant(...)* of *DBMMetaProperty.*
3. The meta-property values are accessible by the methods defined on *DBMGenericObject* class:
   - if the meta-property is mono-valued*: getMetaPropertyValue(meta-property name), getMetaProperty[Type]Value(meta-property name)*, *setMetaPropertyValue(meta-property name, new value)* and *setMetaProperty[Type]Value(meta-property name, new value)* where [Type] = "Boolean", "Char", "Int", "Double" and "String".
   - if the meta-property is multi-valued*: getMetaPropertyListValue(meta-property name), getMetaProperty[Type]ListValue(meta-property name)*, *setMetaPropertyListValue(meta-property name,new value array)* and *setMetaProperty[Type]ListValue(meta-property name,new value array)* where [Type] = "Boolean", "Char", "Int", "Double" and "String".

b) Methods on relationships between classes

The methods to get a class through a link between two classes are called:

- *getFirstClass()*, *getLastClass()*, *getNextClass(previous object)* and *getPreviousClass(next object)* if the class has a collection of objects (i.e. the class plays a role 0-N in the relationship between the two classes).
  Examples (see figure 3.1 in section 3.1.2): *getFirstProduct()*, *getLastProduct(), getNextProduct(...)* and *getPreviousProduct(...)* of *DBMProject*.
- *getClass()* if the class is linked to only one object (i.e. the class plays a role 1-1 or 0-1 in the relationship between the two classes).
  Example (see figure 3.1 in section 3.1.2): *getProject()* of *DBMProduct*.
- if the accessed class has subtypes, *get* methods are also defined on the subtypes. In this case, subtype class names are added at the end of the get method names.
  Examples (see figure 3.1 in section 3.1.2): *getFirstProductSchema()*, *getLastProductSchema()*, *getNextProductSchema(...)* and *getPreviousProductSchema(...)* of *DBMProject*.

- To avoid ambiguity when classes are linked by more than one rel-type, the relationship name is also used in the *get* method name.
  Example (see figure 3.1 in section 3.1.2): *getFirstToConnection()*, *getLastToConnection()*, *getNextFromConnection(…)* and *getPreviousFromConnection(…)* of *DBMProduct*.
- Because multi-valued compound attributes can have groups and belong to groups, the methods to get the groups that contain a specific component are called *getFirstMemberOfGroup()*, *getLast-MemberOfGroup()*, *getNextMemberOfGroup(…)* and *getPreviousMemberOfGroup(…)*. This is an exception to the previous rules in order to avoid ambiguity.

The methods to add a link between two classes are called:

- *addFirstClass(current object)* and *addNextClass(current object, previous object)* if the class has a collection of object (i.e. the class plays a role 0-N in the relationship between the two classes).
  Examples (see figure 3.1 in section 3.1.2): *addFirstProduct(…)* and *addNextProduct(…)* of *DBMProject*.
- Methods called addClass() are only defined if the parent class plays a role 1-1 or 0-1 in a one-to-one (all roles have a maximum cardinality of 1) relationship.
  Examples (figure 3.3 in section 3.1.4): addProcessingUnitCardinality(…) of DBMConsumption.
- To avoid ambiguity when classes are linked by more than one rel-type, the relationship name is also used in the add method name.
  Examples (see figure 3.1 in section 3.1.2): addFirstToConnection (…) and addNextFromConnection (…) of DBMProduct.

The methods to remove links between two classes are called *removeClass()*. These methods are defined on each side of the rel-types. Their names follow same rules as *get* methods except that *remove* methods are not propagated to the subtypes.

Examples: *removeProduct()*, *removeFromConnection()*, *removeToConnection()* and *removeMemberOfGroup()*. Method *removeProductSchema()* does not exist.

c) Methods on classes

Methods to create a new object are called *createClass(parameters)* or *createUniqueClass(parameters)* where parameters describe all the fields of the new object and the object linked by a mandatory 1-1 role. Sometimes parameters also contain the previous object in the collection list of the parent object. *createUniqueClass* methods ensure that the name of the created object is unique in the repository (suffix must be added). These methods are defined on the parent class (playing a 0-N role in the rel-type) and only exist for final subtypes. Note that, in the DB-MAIN repository, project and meta-object cannot be created.

Example (see figure 3.1 in section 3.1.2): *createSchema(String name, String short_name, String version, Date creation_date, Date last_update, char type, DBMSchema prev)* of *DBMProject*.

Methods to delete an object are called *deleteClass()* and are defined on the class and any subtypes.

Examples (see figure 3.1 in section 3.1.2): *deleteProduct()* of *DBMProduct* and *deleteSchema()* of *DBMSchema*.

Methods to transform an object are called *transfoInto...()* (or something like that) and are defined on the objects that can be transformed.

Examples (see figure 3.2 in section 3.1.3): *transformIntoEntityType()* of *DBMAttribute*.

Methods to copy a given object are called *copyClass(parameters)* where parameters describe the original object and the previous object in the collection list of the parent object. These methods are defined on the parent class (playing a 0-N role in the rel-type) and only exist for final subtypes. Note that the

copySchema method is defined on the DBMSchema class and the parameters are name, version and process name (name of the process in the history).

Examples (see figure 3.1 in section 3.1.2): *copyEntityType(DBMEntityType ent, DBMEntityType prev)* of *DBMSchema*.

3.1.2 The project view

Each DB-MAIN repository describes all the specifications related to a project as well as the activities that were carried out to produce these specifications. A logical piece of specification appears as a product. This section analyzes the concepts of project, product and meta-definition (figure 3.1).

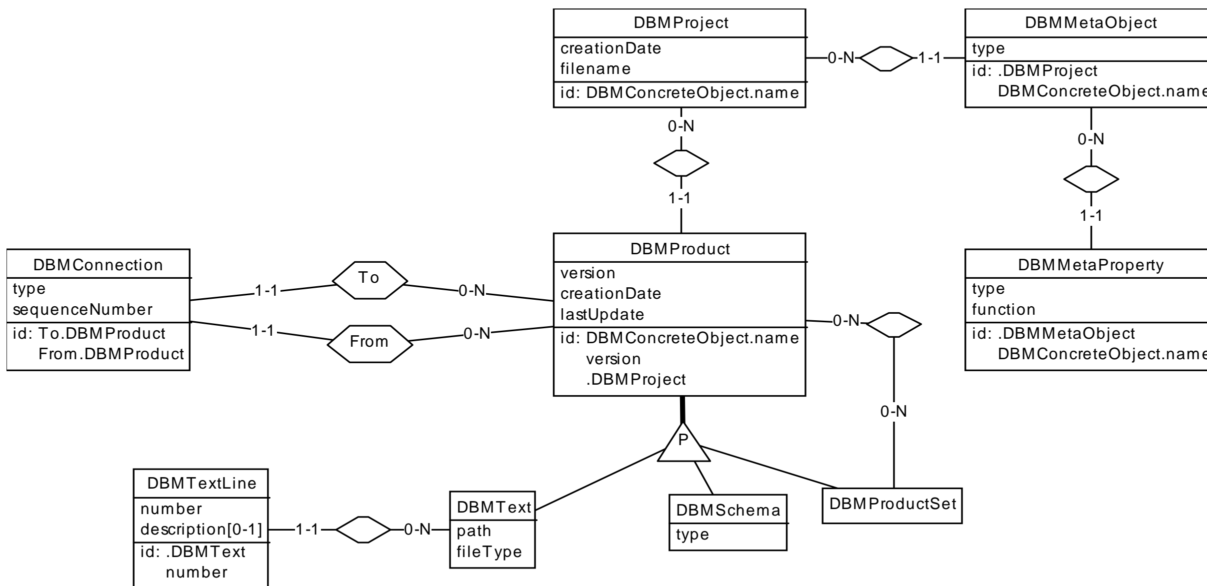


Figure 3.1 - The project view of the DB-Main repository for JIDBM.

3.1.2.1 DBMProject

A *DBMProject* object represents the project in DB-MAIN repository. There is only one instance of this object in DB-MAIN. The JIDBM library does not allow programmers to create a new project. To achieve that, the project must be created in DB-MAIN (Menu File/New Project…). Once the project is loaded or created in DB-MAIN, to access it, an instance of the object DBMProject is created by:

```
DBMProject pro = new DBMProject();
```

The *DBMProject* class has the field *creationDate*. *DBMProject* inherits properties of *DBMConcreteObject*. This class contains fields *name*, *shortName*, *semanticDescription* and *technicalDescription*. A *DBMProject* instance is identified by its name.

A project is made up of a collection of products (*DBMProduct*). The products fall into three classes: data and process schemas (*DBMSchema*), files (*DBMText*) and product sets (*DBMProductSet*). A project also has a collection of meta-objects (*DBMMetaObject*).

3.1.2.2 DBMMetaObject and DBMMetaProperty

The repository part made up of the *DBMMetaObject* and *DBMMetaProperty* classes are called the meta-definition.

A *DBMMetaObject* object represents the meta-objects of the DB-MAIN repository. The meta-objects are predefined in the repository. They describe the main object types in DB-MAIN and allow dynamic properties to be defined. The JIDBM interface does not allow creating or modifying meta-objects. A meta-object has a *name*, a *semanticDescription* (from super-type *DBMConcreteObject*) and a *type*. It is identified by its name and project. The predefined DBMMetaObjects are:

DBMMetaObject		Corresponding JIDBM Class
name	**type**	
Project	DBMGenericObject.SYSTEM	DBMProject
Schema	DBMGenericObject.SCHEMA	DBMSchema
Text	DBMGenericObject.TEXT	DBMText
Product set	DBMGenericObject.PROD_SET	DBMProductSet
Entity type	DBMGenericObject.ENTITY_TYPE	DBMEntityType
Rel-type	DBMGenericObject.REL_TYPE	DBMRelationshipType
Atomic attribute	DBMGenericObject.SI_ATTRIBUTE	DBMSimpleAttribute
Compound attribute	DBMGenericObject.CO_ATTRIBUTE	DBMCompoundAttribute
Group	DBMGenericObject.GROUP	DBMGroup
Role	DBMGenericObject.ROLE	DBMRole
Collection	DBMGenericObject.COLLECTION	DBMCollection
Processing unit	DBMGenericObject.PROC_UNIT	DBMProcessingUnit
Processing unit relation	DBMGenericObject.REL_ELEMENT	DBMElementRelation
Association	DBMGenericObject.CONSUMPTION	DBMConsumption
Use case association role	DBMGenericObject.CONS_PU	DBMProcessingUnitCardinality
Actor association role	DBMGenericObject.CONS_RES	DBMResourceCardinality
Actor	DBMGenericObject.RESOURCE	DBMResource
Actor generalization	DBMGenericObject.RE_ISA	DBMResourceSubType
State	DBMGenericObject.STATE	DBMState
In-out	DBMGenericObject.ENVIRONMENT	DBMEnvironment

Table 3.1 - The predefined meta-objects in DB-MAIN repository.

A meta-object has a collection of meta-properties (*DBMMetaProperty*) and belongs to a project.

DBMMetaProperty represents a dynamic property of a meta-object. To each meta-object predefined in DB-MAIN, some meta-properties are created automatically. The user can manage all of them and create new ones to DB-MAIN meta-objects. The new meta-properties will be managed by DB-MAIN. A dynamic property is a new property for the corresponding DB-MAIN repository objects.

A meta-property has a *name*, *semanticDescription* (from super-type *DBMConcreteObject*), a *type* and a *function*. It is identified by its name and meta-object. The following table describes the possible values of the fields *type* and *function*.

Field type of DBMMetaProperty class	
Constant	**Description**
DBMMetaProperty.BOO_ATT	Boolean type
DBMMetaProperty.CHAR_ATT	Character type
DBMMetaProperty.CHAR_ATT	Real type
DBMMetaProperty.NUM_ATT	Integer type
DBMMetaProperty.VARCHAR_ATT	String type
Field function of DBMMetaProperty class	
Constant	**Description**
DBMMetaProperty.MODIF_MP	Updatable meta-property
DBMMetaProperty.MULTI_MP	Multi-valued meta-property

Table 3.2 - Constants of the fields *type* and *function* in the *DBMMetaProperty* class.

DBMMetaProperty.PREDEF_MP	Meta-property with predefined values (defined in field semanticDescription)
DBMMetaProperty.SYSTEM_MP	System meta-property

Table 3.2 - Constants of the fields *type* and *function* in the *DBMMetaProperty* class.

For example, the code below creates a new multi-valued and predefined meta-property "Owner" for the object Schema:

```
DBMProject pro = new DBMProject();
// Create the "Owner" meta-property
DBMMetaObject mo = pro.getFirstMetaObject();
while (mo != NULL) {
    if (mo.getType() == DBMGenericObject.SCHEMA) {
        DBMMetaProperty mp = mo.createMetaPproperty(
            "Owner", DBMMetaProperty.VARCHAR_ATT,
            DBMMetaProperty.MULTI_MP & DBMMetaProperty.PREDEF_MP,
            NULL);
        mp.setSemanticDescription("#VALUES=\nJohn\nRichard\nLee\n#");
        break;
    }
    mo = pro.getNextMetaObject(mo);
}
```

The meta-property "Owner" values[2] can be obtained by the following code:

```
DBMProject pro = new DBMProject();
DBMSchema sch = pro.getFirstProductSchema();
while (sch != NULL) {
    Object [] l;
    l = sch.getMetaPropertyListValue("Owner");
    if (l != null) {
       if (l.length > 0) {
           System.out.print("Owner: ");
           int i = 0;
           while (i < l.length) {
               System.out.print(l[i]+" ");
               i = i + 1;
           }
           System.out.print("\n");
       }
    }
    sch = pro.getNextProductSchema(sch);
}
```

3.1.2.3 DBMProduct and DBMConnection

DBMProduct is the super-type of *DBMSchema*, *DBMText* and *DBMProductSet*. *DBMProduct* has fields *name*, *shortName*, *semanticDescription*, *technicalDescription* (inherited from super-type *DBMConcreteObject*), *version*, *creationDate* and *lastUpdate*. An instance of *DBMProduct* is identified by its *name*, *version* and project. A product can be an item of several product sets (see section 3.1.2.6).

DBMConnection represents a connection that establishes an oriented link between product objects (*DBMProduct*) like schemas (*DBMSchema*), documents (*DBMText*) or product sets (*DBMProductSet*). A product can have a collection of connections through the rel-types *To* and *From*. An instance of *DBMConnection* is linked to the origin product by the relationship *From* and to the target product by

2. The method to obtain or modify meta-property values are defined on the *DBMGenericObject* class (see also section 3.1.1.3 for more details).

the relationship *To*. *DBMConnection* has the fields: *type* (see table below for possible values) and *sequenceNumber* (to order connections of a product).

Field type of DBMConnection class	
Constant	**Description**
DBMConnection.CON_COPY	Copy link between two products
DBMConnection.CON_DIC	Report link between a schema and a text
DBMConnection.CON_GEN	Generation link between a schema and a text
DBMConnection.CON_INTEG	Integration link between two schemas
DBMConnection.CON_XTR	Extract link between a text and a schema

Table 3.3 - Constants of the field *type* in the *DBMConnection* class.

3.1.2.4 DBMText and DBMTextLine

DBMText represents a document like files, documentation, programs, DDL[3] code, etc. Since *DBMText* is a subtype of *DBMProduct*, it inherits fields and methods of *DBMProduct*. It also has the fields: *path* (the path of the file) and *fileType* (the table below describes the predefined values). A text can have a collection of text lines.

Field typeFile of DBMText class	
Constant	**Description**
DBMText.FILE_GEN_DIC	Report generation file
DBMText.FILE_GEN_SQL_AC	Academic SQL generation file
DBMText.FILE_GEN_SQL_ACCESS	Access generation file
DBMText.FILE_GEN_SQL_INTERBASE	Interbase generation file
DBMText.FILE_GEN_SQL_MYSQL	MySQL generation file
DBMText.FILE_GEN_SQL_POSTGRESQL	PostgreSQL generation file
DBMText.FILE_GEN_SQL_STD	Standard SQL generation file
DBMText.FILE_XTR_COB	COBOL extraction file
DBMText.FILE_XTR_COD	CODASYL extraction file
DBMText.FILE_XTR_IMS	IMS extraction file
DBMText.FILE_XTR_SQL	SQL extraction file

Table 3.4 - Constants of the field *typeFile* in the *DBMText* class.

DBMTextLine represents a text line. This class has the field: *number* (the line number in the corresponding text) and *description* (a free text on the line). This class allows annotating lines of a text represented by an instance of the class *DBMText*.

3.1.2.5 DBMSchema

DBMSchema represents a generalized data or process schema. A schema belongs to only one *DBMProject* and is identified by its name and version.

Since *DBMSchema* is a subtype of *DBMProduct*, it inherits properties and methods of *DBMProduct*. It also has the field: *type* (table below describes possible values).

field type of DBMSchema class	
Constant	**Description**
DBMSchema.ERASCHEMA	Entity-Relationship schema
DBMSchema.UMLACTIVITY	UML activity diagram
DBMSchema.UMLCLASSSCHEMA	UML class diagram
DBMSchema.UMLUSECASE	UML use case diagram

Table 3.5 - Constants of the field *type* in the *DBMSchema* class.

3. Data Definition Language.

3.1.2.6 DBMProductSet

DBMProductSet represents a set of products that can contain any number of products. A product set can contain a collection of products. An integrity constraint is that a set cannot contain itself.

Since *DBMProductSet* is a subtype of *DBMProduct*, it inherits properties and methods of *DBMProduct*.

3.1.3 The data view

The data view (figure 3.2) models data schemas. It mainly comprises entity types, rel-types, attributes, domains, collections, anchored processing units and various constraints (expressed as properties of groups of components).

Figure 3.2 - The data view of the DB-Main repository for JIDBM.

3.1.3.1 DBMSchema

A data schema mainly consists of entity types (*DBMEntityType*), rel-types (*DBMRelationshipType*) and collections (*DBMCollection*). A schema can own several data objects (*DBMDataObject*) and collections (*DBMCollection*). It has two interfaces that implement attribute (*DBMAttributeOwner*) and processing unit (*DBMProcessingUnitOwner*) owners. This part of JIDBM repository is used to manage ER schemas and UML class diagrams. An UML class diagram is the same as an ER schema. Only the graphical view of these schemas is different. In particular, the roles of a rel-type are stored in the repository in the same way for UML class diagrams as for ER schemas, even if they are inverted on screen. Note that non-binary rel-types are difficult to manage in a UML class diagram, so they should be avoided, even if they are permitted.

3.1.3.2 DBMCollection

DBMCollection denotes a collection that is a repository for data objects. In fact, in a data schema, DB-MAIN only supports entity types in collections. An instance of collection can comprise entities from

different entity types, and the entities of a given type can be stored in several collections. A collection has fields *name*, *shortName*, *semanticDescription* and *technicalDescription* (inherited from *DBMConcreteObject*). It always belongs to only one schema and is identified by its name and schema. A collection can contain several data objects (only entity types).

3.1.3.3 DBMDataObject

DBMDataObject is a generalization of objects denoting entity types (*DBMEntityType*), rel-types (*DBMRelationshipType*), attributes (*DBMAttribute*) and processing units (*DBMProcessingUnit*). *DBMDataObject* is the main object of a schema. A data object has fields *name*, *shortName*, *semanticDescription* and *technicalDescription* (inherited from *DBMConcreteObject*). It can belong to several collections (see section 3.1.3.2), can have several groups (see section 3.1.3.11) and can be the domain of several simple attributes (see section 3.1.3.8). A data object always belongs to only one schema.

3.1.3.4 DBMEntityRelationshipType

DBMEntityRelationshipType instances are generalizations of entity types (*DBMEntityType*) and rel-types (*DBMRelType*). Since *DBMEntityRelationshipType* is a subtype of *DBMDataObject*, it inherits all the properties and methods of *DBMDataObject*. A entity-rel-type can be owner of attributes (through interface *DBMAttributeOwner*, see section 3.1.3.12) and processing units (through interface *DBMProcessingUnit*, see section 3.1.3.13).

3.1.3.5 DBMEntityType, DBMCluster and DBMSubType

DBMEntityType denotes an entity type (also called class in UML class diagram) of data schemas. Since *DBMEntityType* is a subtype of *DBMEntityRelationshipType*, it inherits all its properties and methods. Using the inheritance mechanism, entity types can belong to collections, have attributes and processing units. It can play roles in rel-types (see section 3.1.3.6), and be generalization/specialization of other entity types. An instance of *DBMEntityType* is identified by its name and schema.

The repository allows an entity-type to be specialised in several ways (e.g. persons divided in male/female or children/adult). To do so, an entity type can be linked to several clusters (*DBMCluster*), one for the gender and one for the age in the example. These clusters can have several subtypes (*DBMSubType*), with values "male" and "female" for the cluster gender, "adult" and "children" for the cluster age. But, in practice, DB-MAIN does not support multiple clusters, so each entity type should not be linked to more than one cluster.

DBMCluster denotes a specialization for an entity type. A cluster has fields *clusterIdentifier*, *type* and *criterion*. The *clusterIdentifier* field is a string to identify the cluster among others of the same entity type. The field *type* can have the values *DBMCLuster.DISJOINT_CLU* (the subtype collection is declared disjoint), *DBMCLuster.TOTAL_CLU* (the subtype collection is declared total) or the combining of two previous values (the subtype collection is a partition). The *criterion* field is a string representing the selection criterion. A cluster must have one or more subtypes.

DBMSubType denotes the possible subtypes of a cluster. A subtype has a *value* (the criterion value to specialize the entity types). It is identified by its value, cluster and entity type.

3.1.3.6 DBMRelationshipType and DBMRole

DBMRelationshipType denotes a rel-type (also called association in UML class diagram) of data schemas. Since *DBMRelationshipType* is a subtype of *DBMEntityRelationshipType*, it inherits all the fields and methods of *DBMEntityRelationshipType*. A rel-type may own attributes, processing units and groups in the same way as entity types. Each rel-type is attached to exactly one schema and identified by its name and schema. A rel-type can have a collection of roles.

DBMRole denotes a role of rel-type. A role is characterized by fields *name*, *shortName*, *semanticDescription*, *technicalDescription technicalDescription* (inherited from *DBMConcreteObject*), *minimumCardinality*, *maximumCardinality* (*DBMRole.N_CARD* constant indicating infinity) and *aggregation*. The values of field *aggregation* are: *DBMRole.AGGREGATION* denotes an UML aggregation association and *DBMRole.COMPOSITION* denotes an UML composition association. A role is always linked to a single rel-type. A role can be played by several entity-types and an entity type can play roles in

several rel-types. A role can be member of several groups (through virtual class *Component*, see section 3.1.3.11). Each role is identified by its name, entity types and the rel-type it depends on.

3.1.3.7 DBMAttribute

DBMAttribute describes the properties (or fields) of attribute owners. It is a generalization of *DBMSimpleAttribute* and *DBMCompoundAttribute*. All the properties of an attribute are shared with subtypes by the inheritance principles. Since *DBMAttribute* is a subtype of *DBMDataObject*, it inherits all the fields and methods of DBMDataObject. An attribute has fields *minimumCardinality*, *maximumCardinality* (*DBMAttribute.N_CARD* constant indicating infinity) and *setType*. The field *setType* defines the set type value for a multi-valued attribute (*maximumCardinality* greater than one). The *setType* possible values are given in the table below.

Field setType of DBMAttribute	
Constant	**Description**
DBMAttribute .ARRAY_ATT	Array type: indexed sequence of cells that can each contain an element
DBMAttribute .BAG_ATT	Bag type: unstructured collection of elements
DBMAttribute .LIST_ATT	List type: sequenced collection of elements
DBMAttribute .SET_ATT	Set type: unstructured collection of distinct elements
DBMAttribute .UARRAY_ATT	Unique array type: indexed sequence of cells that can each contain a distinct element
DBMAttribute .ULIST_ATT	Unique list type: sequenced collection of distinct elements

Table 3.6 - Constants of the field *setType* in the *DBMAttribute* class.

An attribute can be member of several groups (through virtual class *Component*, see section 3.1.3.11). It belongs to only one owner of attribute (represented by *DBMAttributeOwner* class, see section 3.1.3.12). The *DBMAttributeOwner* class is a interface in the JIDBM library. It is a abstract class which is a generalization of the attribute's parent (entity type, rel-type or compound attribute). An attribute is identified by its name and owner.

3.1.3.8 DBMSimpleAttribute

DBMSimpleAttribute represents simple attributes (i.e. attributes with atomic type). Since DBMSimpleAttribute is a subtype of *DBMAttribute*, it inherits all the properties and methods of *DBMAttribute*. A simple attribute has fields *type*, *stable* (value cannot be changed) and *recyclable* (value can be reused) properties, *length* (total length) and *decimalNumber* (number of decimals among the total length). The possible values of the field *type* are:

Field type of DBMSimpleAttribute	
Constant	**Description**
DBMSimpleAttribute.BOO_ATT	Boolean type
DBMSimpleAttribute.CHAR_ATT	Character type
DBMSimpleAttribute.DATE_ATT	Date type
DBMSimpleAttribute.FLOAT_ATT	Real type
DBMSimpleAttribute.INDEX_ATT	Index type
DBMSimpleAttribute.NUM_ATT	Integer type
DBMSimpleAttribute.OBJECT_ATT	Object type
DBMSimpleAttribute.SEQ_ATT	Sequence type
DBMSimpleAttribute.USER_ATT	User-defined type
DBMSimpleAttribute.VARCHAR_ATT	String type

Table 3.7 - Constants of the field *type* in the *DBMSimpleAttribute* class.

Object type attributes are represented by *DBMSimpleAttribute* objects whose field *type* equals constant *DBMSimpleAttribute.OBJECT_ATT* and which are connected to a data object. This data object must be an entity type. User-defined attributes are represented by *DBMSimpleAttribute* objects whose field *type*

equals constant *DBMSimpleAttribute.USER_ATT* and which are connected to a data object. This data object must be an attribute. This attribute must belong to the special entity type "*]DOMAINS-ATTRIBUTES[*" which belongs to the special schema named "*]DOMAINS[*" (with version "*]USER-DEFINED[*").All other attributes are represented by *DBMSimpleAttribute* instances not linked to a data object. The field *type* identifies the type of the attribute.

3.1.3.9 DBMCompoundAttribute

DBMCompoundAttribute describes compound attributes. Since *DBMCompoundAttribute* is a subtype of *DBMAttribute*, it inherits all the properties and methods of *DBMAttribute*. A compound attribute may own attributes and groups in the same way as entity types and rel-types.

3.1.3.10 DBMProcessingUnit

In data schemas, *DBMProcessingUnit* describes processing units (method, procedure, trigger or predicate) anchored either to a schema, an entity type or a rel-type. For representing a more precise definition of a processing unit, the DB-MAIN user will use a UML activity or use case diagram (see section 3.1.4.3). Since *DBMProcessingUnit* is a subtype of *DBMDataObject*, it inherits all the properties and methods of *DBMDataObject*. A processing unit has fields *type* and *mode*. The field *mode* is only used in process schema (see section 3.1.4.3).The field *type* defines the processing unit type which the possible values are:

Field type of DBMProcessingUnit	
Constant	**Description**
DBMProcessingUnit.METHOD_TYPE_PU	Method type
DBMProcessingUnit.PREDICAT_TYPE_PU	Predicat type
DBMProcessingUnit.PROCEDURE_TYPE_PU	Procedure type
DBMProcessingUnit.TRIGGER_TYPE_PU	Trigger type

Table 3.8 - Constants of the field *type* in the *DBMProcessingUnit* class

A processing unit belongs to only one owner (represented by *DBMProcessingUnitOwner* class see section 3.1.3.13). The *DBMProcessingUnitOwner* class is an interface in the JIDBM library. It is a abstract class which is a generalization of the parent (entity type, rel-type or schema) of a processing unit. A processing unit is identified by its name and owner.

3.1.3.11 DBMGroup, DBMConstraint and DBMConstraintMember

DBMGroup describes a group made up of attributes, roles or other groups. A group represents a construct attached to an entity type, a rel-type or a multi-valued compound attribute. It is used to represent concepts such as identifiers, foreign keys, indexes, sets of exclusive or coexistent attributes … A entity type group can comprise inherited attributes and roles, i.e., components from its direct or indirect supertypes.

A group has fields *name*, *semanticDescription*, *technicalDescription* (inherited from *DBMConcreteObject*), *type*, *function*, *minimumCardinality* and *maximumCardinality* (*DBMGroup.N_CARD* constant indicating infinity). The field *type* is not used and the field *function* (defining the group functions) can have the values in the table below. A group can have many values for field *function* (e.g. DBMGroup.PRIM_GR and DBMGroup.KEY_GR).

Field function of DBMGroup	
Constant	**Description**
DBMGroup.AL1_GR	At-least-one constraint
DBMGroup.COEX_GR	Coexistence constraint
DBMGroup.CST_GR	Generic user-defined constraint
DBMGroup.EXCL_GR	Exclusion constraint
DBMGroup.ID_GR	Identifier (primary or secondary)

Table 3.9 - Constants of the field *type* in the *DBMGroup* class.

DBMGroup.KEY_GR	Access key constraint (index)
DBMGroup.NONE_GR	No constraint
DBMGroup.PRIM_GR	Primary identifier
DBMGroup.SEC_GR	Secondary identifier

Table 3.9 - Constants of the field *type* in the *DBMGroup* class.

A group belongs to a data object (entity type, rel-type or multi-valued compound attribute). It is identified by its name and data object. It can have a collection of components (represented by *Component* class). The *Component* class does not exist in the JIDBM library. It is a virtual class which is a generalization of the members (attributes, roles or other groups) of a group. Note that, in an entity type, the roles that can be part of a group are the "far" roles of the rel-types into which the entity type participates. A group can be the support of several constraint members.

An inter-group constraint (like a referential constraint between a reference group and an identifier) is represented by an instance of class *DBMConstraint*. A constraint has a field *type* for which the possible values are:

Field type of DBMConstraint	
Constant	**Description**
DBMConstraint.EQ_CONSTRAINT	Equality constraint
DBMConstraint.GEN_CONSTRAINT	Generic user-defined constraint
DBMConstraint.INC_CONSTRAINT	Inclusion constraint
DBMConstraint.INV_CONSTRAINT	Inverse constraint
DBMConstraint.REF_CONSTRAINT	Referential constraint

Table 3.10 - Constants of the field *type* in the *DBMConstraint* class.

A constraint is linked to one or more group through the *DBMConstraintMember* class representing the constraint member. A member constraint has a *memberRole* field which can have the following predefined values:

Field memberRole of DBMConstraintMember	
Constant	**Description**
DBMConstraintMember.OR_MEM_CST	Constraint origin
DBMConstraintMember.TAR_MEM_CST	Constraint target

Table 3.11 - Constants of the field *memberRole* in the *DBMConstraintMember* class.

A constraint member materializes the link between a constraint and groups implied in this constraint. It is identified by its constraint and group. DB-MAIN presently supports constraints between two groups only. The origin group of the constraint should be linked to a constraint member with the field *memberRole* sets to the constant *DBMConstraintMember.OR_MEM_CST*, and the target group should be linked to a member constraint with the field *memberRole* sets to the constant *DBMConstraintMember.TAR_MEM_CST*. These two constraint members must be linked to the same constraint.

3.1.3.12 DBMAttributeOwner

DBMAttributeOwner is an interface for attribute owners (*DBMCompoundAttribute* and *DBMEntityRelationshipType*). An interface is an abstract type that is used to specify an interface (in the generic sense of the term) that classes must implement. It is used to encode similarities (methods to manage attributes: create, get, copy, remove, transform, ...) which classes of various types share.

3.1.3.13 DBMProcessingUnitOwner

DBMProcessingUnitOwner is an interface for processing unit owners (*DBMSchema* and *DBMEntityRelationshipType*). An interface is an abstract type that is used to specify an interface (in the generic sense of the term) that classes must implement. It is used to encode similarities (methods to manage processing units: create, get, copy, remove, ...) which classes of various types share.

3.1.4 The process view

Independent processing units such as program, procedures, activities or use cases need being defined in specific products, namely the processing schemas. A processing view (figure 3.3) models processing schemas including action states, internal objects, external objects, states, use cases, actors and relations.

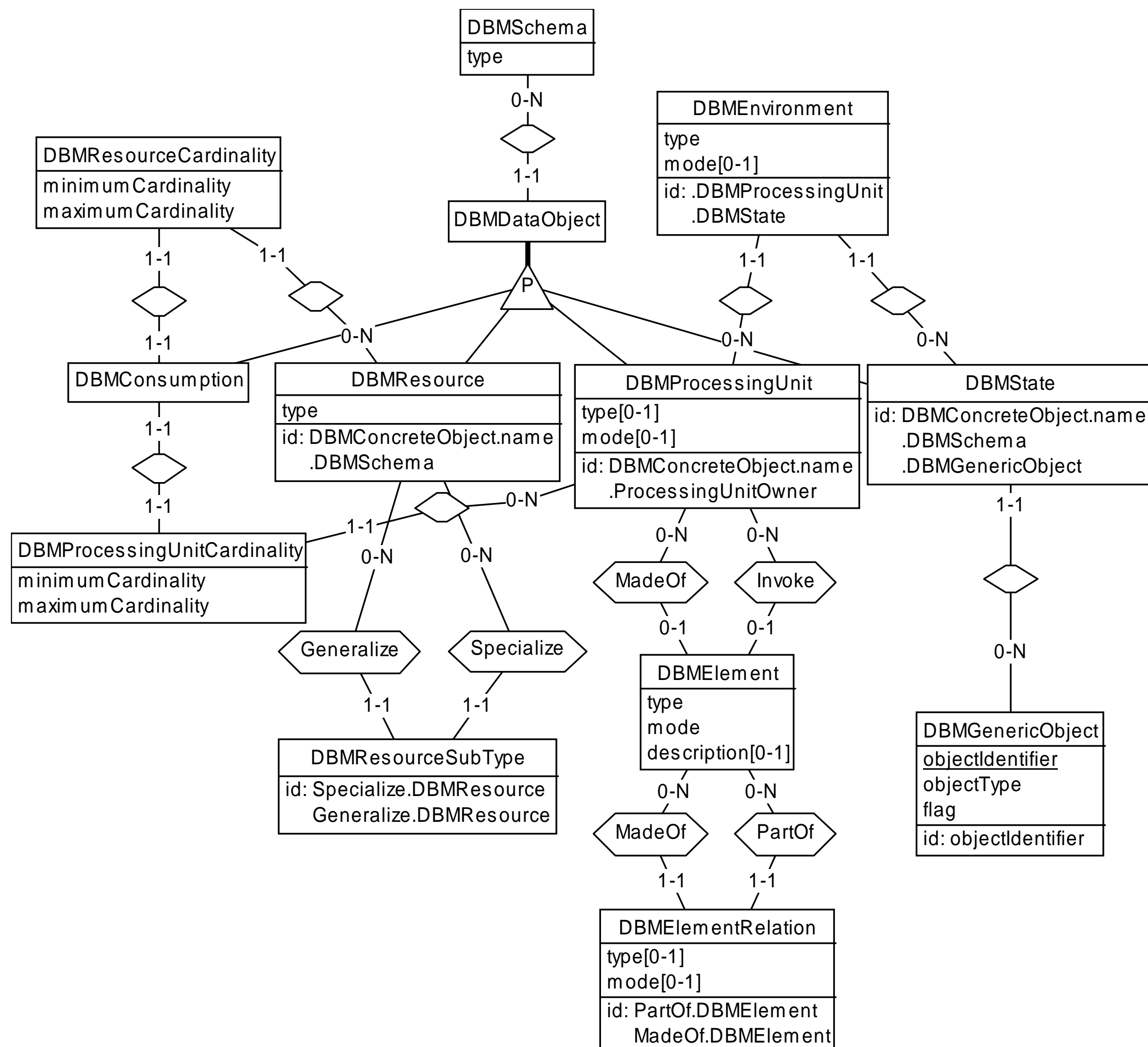


Figure 3.3 - The process view of the DB-MAIN repository for JIDBM.

3.1.4.1 DBMSchema

A process schema mainly consists of processing units (*DBMProcessingUnit*), states (*DBMState*) and resources (*DBMResource*). A schema can own several data objects (*DBMDataObject*). This part of JIDBM repository is used to manage UML activity and use case diagrams. A UML activity diagram (field *type* equals constant *DBMSchema.UMLACTIVITY_DIAGRAM*) is made up of processing units, with various roles and various graphical representations, object states and relations between all these components. A UML use case diagram (field *type* equals constant *DBMSchema.UMLUSECASE_DIAGRAM*) is made up of use cases (represented by processing units), actors (represented by resources) and relations between these objects.

3.1.4.2 DBMDataObject

DBMDataObject is a generalization of objects denoting processing units (*DBMProcessingUnit*, states (*DBMState*), resources (*DBMResource*) and consumptions (*DBMConsumption*). *DBMDataObject* is the main object of a process schema. A data object has the fields *name*, *shortName*, *semanticDescription* and *technicalDescription* (inherited from *DBMConcreteObject*). A data object always belongs to one schema only.

3.1.4.3 DBMProcessingUnit

DBMProcessingUnit describes action states (in activity diagrams) or use cases (in use case diagrams). Since *DBMProcessingUnit* is a subtype of *DBMDataObject*, it inherits all the properties and methods of *DBMDataObject*. A processing unit has the fields *type* and *mode*. The field *type* can be used for documentation (see section 3.1.3.10 for more details). The field *mode* defines the kind of action states. Its possible values are:

Field mode of DBMProcessingUnit	
Constant	**Description**
DBMProcessingUnit.DECISION_MODE_PU	Decision state
DBMProcessingUnit.FINAL_MODE_PU	Final state
DBMProcessingUnit.HOR_SYNC_MODE_PU	Horizontal synchronisation bar
DBMProcessingUnit.INITIAL_MODE_PU	Initial state
DBMProcessingUnit.SIGNAL_REC_MODE_PU	Signal receipt
DBMProcessingUnit.SIGNAL_SEND_MODE_PU	Signal sending
DBMProcessingUnit.VER_SYNC_MODE_PU	Vertical synchronisation bar

Table 3.12 - Constants of the field *mode* in the *DBMProcessingUnit* class.

The fields *type* and *mode* are not used when a processing unit describes a use case.

A processing unit also inherits roles played by data objects with groups and simple attributes, but it never uses them. Using them can lead to unexpected behaviour of DB-MAIN, possibly to crash. A processing unit belongs to only one owner (represented by *DBMProcessingUnitOwner* class). This owner is always the schema. In process schema, a processing unit cannot be attached to entity types or rel-types.

A processing unit can also have several environments representing object flows with states (see section 3.1.4.6). It can also be origin or target of elements (see section 3.1.4.5). A processing unit is identified by its name and schema.

3.1.4.4 DBMState

DBMState represents a particular state of a data object at a given time. A data object is either an internal attribute, or an external attribute, entity type, rel-type or collection. Internal means that the attribute is part of the schema, and external means that the data object is in fact a reference to a data object defined in a data schema (ER schema or UML class diagram). Since *DBMState* is a subtype of *DBMDataObject*, it inherits all its properties and methods. A state also inherits roles played by data objects with groups and simple attributes, but it never uses them. Using them can lead to unexpected behaviour of DB-MAIN, possibly to crash.

An object state is always linked to a generic object. This generic object must be the super-type of:

An internal attribute that must belong to the special entity type "*VARIABLES*" which belongs to the same schema. This attribute must be simple or compound but does not participate to groups.

An external data object (attribute, entity type, rel-type or collection) defined into a data schema.

A state can also have several environments representing object flows with processing units (see section 3.1.4.6). It is identified by its name, schema and generic object.

3.1.4.5 DBMElement and DBMElementRelation

A control flow between two action states (processing units in activity diagram) and an extend, include or generalization relation between two use cases (processing units in use case diagram) are represented by an instance of *DBMElementRelation* class, and two instances of *DBMElement* class.

DBMElement describes an element. An element has fields *type*, *mode* (not used) and *description* (not used). It is linked to an origin (through *MadeOf* relationship) or target (through *Invoke* relationship) processing unit.

DBMElementRelation describes an element relation. An element relation has fields *name, semanticDescription*, *technicalDescription* (inherited from *DBMConcreteObject*), *type* (not used) and *mode* (not used). It is linked to an origin (through *MadeOf* relationship) and a target processing units (through *PartOf* relationship). It is identified by its two elements.

Elements and element relations are used to link processing units. An element is linked with *MadeOf* to the processing unit from which the control flow is originating. Another element is linked with *Invoke* to the processing unit to which the control flow is targeted. These two elements are linked by an element relation whose type and mode fields can be left undefined. Note that each originating processing unit is linked to at most one element with the same type value which is shared by all the relations of the same kind originating from this processing unit. The field *type* of the two elements should have the following values, according to the kind of relation:

Field type of DBMElement	
Constant	**Description**
DBMElement.CTRL_SET_ELEM	Origin of control flow
DBMElement.CTRL_TYPE_ELEM	Target of control flow
DBMElement.EXTEND_SET_ELEM	Origin of entend relationship
DBMElement.EXTEND_TYPE_ELEM	Target of entend relationship
DBMElement.GEN_SET_ELEM	Origin of generalization relationship
DBMElement.GEN_TYPE_ELEM	Target of generalization relationship
DBMElement.INCLUDE_SET_ELEM	Origin of include relationship
DBMElement.INCLUDE_TYPE_ELEM	Target of include relationship

Table 3.13 - Constants of the field *type* in the *DBMElement* class.

3.1.4.6 DBMEnvironment

In an activity diagram, an object flow is represented by an instance of *DBMEnvironment* class. *DBMEnvironment* describes an environment. An environment has fields *name, semanticDescription*, *technicalDescription* (inherited from *DBMConcreteObject*), *type* and *mode*. It is linked to a processing unit (action state) and to an object state. The orientation of the link is stored in the *mode* field. The *type* field reminds whether the object, whose state is linked, is internal or external. The possible values of these two fields are:

Field type of DBMEnvironment	
Constant	**Description**
DBMEnvironment.EXT_TYPE_PENV	Object flow with a data object of another schema
DBMEnvironment.INT_TYPE_PENV	Object flow with a data object of the same schema
Field mode of DBMEnvironment	
DBMEnvironment.IN_MODE_PENV	Input object flow
DBMEnvironment.OUT_MODE_PENV	Output object flow
DBMEnvironment.UPD_MODE_PENV	Update object flow

Table 3.14 - Constants of the fields *type* and *mode* in the *DBMEnvironment* class.

An environment is identified by its state and processing unit.

3.1.4.7 DBMResource and DBMResourceSubType

In a use case diagram, *DBMResource* represents actors. Since *DBMResource* is a subtype of *DBMDataObject*, it inherits all the properties and methods of *DBMDataObject*. A resource has also got a *type* (not used) field. It also inherits roles played by data objects with groups and simple attributes, but it never uses them. Using them can lead to unexpected behaviour of DB-MAIN, possibly to crash. A resource can be structured as a hierarchy through *Specialize* and *Generalize* relationship. It can also play several roles in association with processing units (see section 3.1.4.8). A resource is identified by its name and schema.

DBMResourceSubType represents an actor generalization in a use case diagram. A resource subtype has fields *name, semanticDescription*, *technicalDescription* (inherited from *DBMConcreteObject*). It is linked to the most general resource with *Generalize* relationship, and to the specialized resource with *Specialize* relationship. It is identified by its subtype and super-type resources.

3.1.4.8 DBMConsumption, DBMProcessingUnitCardinality and DBMResourceCardinality

In a use case diagram, an association between a use case (processing unit) and an actor (resource) is represented by an instance of *DBMConsumption* class. Since *DBMConsumption* is a subtype of *DBMDataObject*, it inherits all the properties and methods of *DBMDataObject*. A consumption also inherits the roles played by data objects with groups and simple attributes, but it never uses them. Using them can lead to unexpected behaviour of DB-MAIN, possibly to crash. A consumption is linked to a processing unit cardinality object and to a resource cardinality object. It is identified by its name and schema.

DBMProcessingUnitCardinality represents the cardinalities of a use case (processing unit) in an association (consumption). A processing unit cardinality has fields *name, semanticDescription*, *technicalDescription* (inherited from *DBMConcreteObject*), *minimumCardinality* and *maximumCardinality* (*DBMProcessingUnitCardinality.N_CARD* constant indicating infinity). The *minimumCardinality* and *maximumCardinality* fields show in how many instances of the use case each actor should participate. A processing unit cardinality is linked to one processing unit and one consumption. It is identified by its processing unit and consumption.

DBMResourceCardinality represents the cardinalities of an actor (resource) in an association (consumption). A resource cardinality has fields *name, semanticDescription*, *technicalDescription* (inherited from *DBMConcreteObject*), *minimumCardinality* and *maximumCardinality* (*DBMResourceCardinality.N_CARD* constant indicating infinity). The *minimumCardinality* and *maximumCardinality* fields show in how many actors of the same type should be associated with the use case. A resource cardinality is linked to exactly one resource and one consumption. It is identified by its resource and consumption.

3.1.5 The concrete view

The concrete view (figure 3.4) presents the graphical, description and note information of the DB-MAIN repository objects.

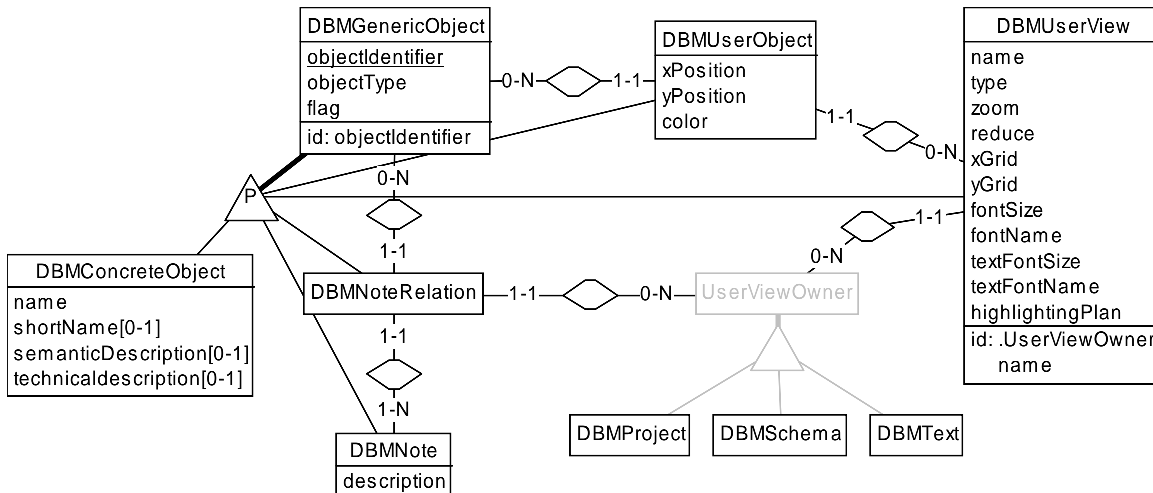


Figure 3.4 - The concrete view of the DB-MAIN repository for JIDBM.

3.1.5.1 DBMGenericObject

DBMGenericObject contains the generic properties of most of DB-MAIN repository objects. In this context, the *DBMGenericObject* class is super-type of most of repository objects (see inheritance view section 3.1.6). A generic object has fields *objectIdentifier* (global identifier for the project; automati-

cally generated; not updatable), *objectType* (a number indicating the object type such as entity type, simple attribute, …; not updatable) and *flag* (indicators dependent on field *objectType*). The predefined constant values of fields *objectType* and *flag* are:

Field objectType of DBMGenericObject	
Constant	**Description**
DBMGenericObject.CLUSTER	Cluster
DBMGenericObject.CO_ATTRIBUTE	Compound attribute
DBMGenericObject.COLLECTION	Collection
DBMGenericObject.CONNECTION	Connection
DBMGenericObject.CONS_PU	Processing unit cardinality
DBMGenericObject.CONS_RES	Resource cardinality
DBMGenericObject.CONSTRAINT	Constraint
DBMGenericObject.CONSUMPTION	Consumption
DBMGenericObject.ELEMENT	Element
DBMGenericObject.ENTITY_TYPE	Entity type
DBMGenericObject.ENVIRONMENT	Environment
DBMGenericObject.GROUP	Group
DBMGenericObject.MEMBER_CST	Constraint member
DBMGenericObject.META_OBJECT	Meta-object
DBMGenericObject.META_PROPERTY	Meta-property
DBMGenericObject.NN_NOTE	Note relation
DBMGenericObject.NOTE	Note
DBMGenericObject.PROC_UNIT	Processing unit
DBMGenericObject.PROD_SET	Product set
DBMGenericObject.RE_ISA	Resource subtype
DBMGenericObject.REL_ELEMENT	Element relation
DBMGenericObject.REL_TYPE	Rel-type
DBMGenericObject.RESOURCE	Resource
DBMGenericObject.ROLE	Role
DBMGenericObject.SCHEMA	Schema
DBMGenericObject.SI_ATTRIBUTE	Simple attribute
DBMGenericObject.STATE	State
DBMGenericObject.SUB_TYPE	Subtype
DBMGenericObject.SYSTEM	Project
DBMGenericObject.TEXT	Text
DBMGenericObject.TEXT_LINE	Text line
DBMGenericObject.USER_OBJECT	User object
DBMGenericObject.USER_VIEW	User view

Table 3.15 - Constants of the field *objectType* in the *DBMGenericObject* class.

Methods *is[Type]* are defined on class DBMGenericObject to simplify the test on object type. For example, *go.isEntityType()* is the same thing as *go.getObjectType() == DBMGenericObject.ENTITY_TYPE*.

Field flag of DBMGenericObject	
Constant	**Description**
DBMGenericObject.ALLPARENTS	Notes only showed when all owners are shown in graphical views
DBMGenericObject.ETROUND	Entity type rectangles with rounded corners in data schema graphical views

Table 3.16 - Constants of the field *flag* in the *DBMGenericObject* class.

DBMGenericObject.ETSHADOW	Entity type rectangles with shadow in data schema graphical views
DBMGenericObject.ETSQUARE	Entity type rectangles with square corners in data schema graphical
DBMGenericObject.GRAPHCOUPLE	Object movements graphically dependent in product graphical views
DBMGenericObject.HAVESEL	Object with selected objects
DBMGenericObject.HIDEASSOCIATION	Associations hidden in use case schema graphical views
DBMGenericObject.HIDEATT	Attributes hidden in data schema graphical views
DBMGenericObject.HIDECTRLFLOW	Control flows hidden in activity schema graphical views
DBMGenericObject.HIDEEXTEND	Extend relationships hidden in use case schema graphical views
DBMGenericObject.HIDEGENACT	Actor generalizations hidden in use case schema graphical views
DBMGenericObject.HIDEGENUC	Use case generalizations hidden in use case schema graphical views
DBMGenericObject.HIDEGR	Groups hidden in data schema graphical views
DBMGenericObject.HIDEINCLUDE	Include relationships hidden in use case schema graphical views
DBMGenericObject.HIDENEWADD	New schema and add text processes hidden in project graphical views
DBMGenericObject.HIDENOTE	Notes hidden in product graphical views
DBMGenericObject.HIDEOBJFLOW	Object flows hidden in activity schema graphical views
DBMGenericObject.HIDEPRIMID	Primary identifiers hidden in data schema graphical views
DBMGenericObject.HIDEPROD	Products belonging to product sets hidden in project graphical views
DBMGenericObject.HIDEPU	Processing units hidden in data schema graphical views
DBMGenericObject.HIDESTEREOTYPE	Stereotypes hidden in product graphical views
DBMGenericObject.ISASTRAIGHT	Is-a relations with square lines in data schema graphical views
DBMGenericObject.LOCK_USED	Methodology control used in project
DBMGenericObject.MARK1	Object marked in highlighting plan 1
DBMGenericObject.MARK2	Object marked in highlighting plan 2
DBMGenericObject.MARK3	Object marked in highlighting plan 3
DBMGenericObject.MARK4	Object marked in highlighting plan 4
DBMGenericObject.MARK5	Object marked in highlighting plan 5
DBMGenericObject.METH_USED	Methodology used in project
DBMGenericObject.PROCESS_TITLE	Process title used in project views
DBMGenericObject.REJECTED	Product rejected
DBMGenericObject.RTROUND	Rel-type rectangles with rounded corners in data schema graphical views
DBMGenericObject.RTSHADOW	Rel-type hexagons with shadow in data schema graphical views
DBMGenericObject.RTSQUARE	Rel-type rectangles with square corners in data schema graphical views
DBMGenericObject.SELECT	Object selected
DBMGenericObject.SHOWTYPEATT	Attribute type shown in data schema graphical views
DBMGenericObject.UMLCENTER	Name of current role or rel-type shifted towards center in class diagram graphical view
DBMGenericObject.UMLCENTERDOWN	Name of current role or rel-type shifted towards bottom center in class diagram graphical view
DBMGenericObject.UMLCENTERUP	Name of current role or rel-type shifted towards top center in class diagram graphical view
DBMGenericObject.UMLLEFTCENTER	Name of current role or rel-type shifted towards center left in class diagram graphical view

Table 3.16 - Constants of the field *flag* in the *DBMGenericObject* class.

DBMGenericObject.UMLLEFTDOWN	Name of current role or rel-type shifted towards bottom left in class diagram graphical view
DBMGenericObject.UMLLEFTUP	Name of current role or rel-type shifted towards top left in class diagram graphical view
DBMGenericObject.UMLRIGHTCENTER	Name of current role or rel-type shifted towards center right in class diagram graphical view
DBMGenericObject.UMLRIGHTDOWN	Name of current role or rel-type shifted towards bottom right in class diagram graphical view
DBMGenericObject.UMLRIGHTUP	Name of current role or rel-type shifted towards top right in class diagram graphical view
DBMGenericObject.WITHOUT_SELECT	Object without selected object

Table 3.16 - Constants of the field *flag* in the *DBMGenericObject* class.

A generic object has a collection of states (see section 3.1.4.4), note relations (see section 3.1.5.4) and user objects (see section 3.1.5.3). It is identified by its object identifier and project.

3.1.5.2 DBMConcreteObject

DBMConcreteObject represents the concrete properties of repository objects that have names and descriptions. Since *DBMConcreteObject* is a subtype of *DBMGenericObject*, it inherits all the properties and methods of *DBMGenericObject*. A concrete object has a *name*, *shortName*, *semanticDescription* and *technicalDescription* fields. Note that, for some subtypes, fields *shortName*, *semanticDescription* or *technicalDescription* are not available. See the previous sections for more details.

3.1.5.3 DBMUserView and DBMUserObject

DBMUserView represents the textual and graphical attributes of some windows (schema, text or project) in the DB-MAIN CASE tool. A user view has a *name*, *type*, *zoom* (zoom factor), *reduce* (reduce factor), *xGrid* (x position of grid), *yGrid* (y position of grid), *fontSize* (graphical font size), *fontName* (graphical font name), *textFontSize* (textual font size), *textFontName* (textual font name) and *highlightingPlan* fields. The possible values of field *type* are:

Field type of DBMUserView	
Constant	**Description**
DBMUserView.COMPACT	Compact textual view
DBMUserView.EXTENDED	Extended textual view
DBMUserView.GRAPH_COMPACT	Compact graphical view
DBMUserView.GRAPH_DEPENDENCY	Dependency graphical view
DBMUserView.GRAPH_STANDARD	Standard graphical view
DBMUserView.SORTED	Sorted textual view
DBMUserView.STANDARD	Standard textual view

Table 3.17 - Constants of the field *type* in the *DBMUserView* class.

A user view contains several user objects and belongs to only one user viewable (represented by the *UserViewOwner* class). The *UserViewOwner* class does not exist in the JIDBM library. It is a virtual class which is a generalization of the parent (schema, text or project) of a user view. A user view is identified by its name and owner. Note that the JIDBM library does not allow users to manage the structured histories of DB-MAIN. In this context, only the compact view (*DBMUserView.GRAPH_COMPACT*) of the project is accessible and manageable by the JIDBM library.

DBMUserObject represents the graphical properties of generic objects in a user view. A user object has fields *xPosition*, *yPosition* and *color*. The positions (*xPosition* and *yPosition*) are defined in thousandth of millimeter (e.g. a *xPosition* value of 10000 represents 1 centimeter). The concept of user object is important because some objects can appear in many views (like an entity type that appears in a UML class diagram and in a UML activity diagram as an external object). A user object is identified by its generic object and user view.

3.1.5.4 DBMNote and DBMNoteRelation

DBMNote represents textual note in schemas and projects. To each element of a schema or project, can be attached a note. A note has a field *description* (free text). It is attached to only one generic object, although, in practice, DB-MAIN only supports notes on the following object types: schema, entity type, rel-type, simple attribute, compound attribute, role, group, collection, cluster, processing unit, element relation, environment, consumption, processing unit cardinality, resource cardinality, state, resource, text, product set.

A note can appear in one or many views (e.g. if it is linked to an entity type that appears in UML class and activity diagrams). Sometimes the note should be visible in all the views, sometimes not. Moreover, an object may receive several notes, possibly in a single view or in many views. This complex situation is managed by the note relation object (represented by the class *DBMNoteRelation*) which materializes a ternary rel-type between note, generic object to which the note is attached, and the user view owner in which view the note should appear. A note relation is identified by its note, generic object, and user view owner.

3.1.6 The inheritance view

The inheritance view (figure 3.5) is just an overview of all the objects which inherit from the DBMGenericObject class.

Figure 3.5 - The inheritance view of the DB-MAIN repository for JIDBM.

3.2 Special classes

This section talks about special classes of the JIDBM library that are not described in the repository architecture.

3.2.1 DBMLibrary

DBMLibrary contains the native methods used by the classes describing the repository to access the C++ repository (*reposit.dll*) of DB-MAIN.

For most methods of the JIDBM repository classes, it exists a native method with the same name suffixed by "Of" and the object name on which the method is defined. For example, the *DBMAttribute* method "*public int getMinimumCardinality()*" is translated in *DBMLibrary* by "*public native int getMinimumCardinalityOfAttribute(int id)*" where id is the object identifier representing the instance on which the method is applied.

DBMLibrary also offers four interesting methods:

- *executeMenu* that launches in DB-MAIN the menu item identified by *id* (see the JIDBM javadoc for more information about the menu constants).
- *getCurrentSchema* and *getCurrentDBMSchema* that return the current schema in DB-MAIN. *getCurrentSchema* returns an object identifier integer and *getCurrentDBMSchema* returns an instance of DBMSchema class.
- *getCurrentProject* and *getCurrentDBMProject* that return the current project in DB-MAIN (only use with *loadLUN* method, see section 2.3.4 for more information). *getCurrentProject* returns an object identifier integer and *getCurrentDBMProject* returns an instance of DBMProject class.
- *loadLUN* that loads a LUN file in the DB-MAIN repository (see section 2.3.4 for more information).
- *unloadLUN* that unloads the project in the DB-MAIN repository (see section 2.3.4 for more information).

3.2.2 DBMConsole

DBMConsole allow programmers to create a console which traps the "*System.out.println*" instructionsand prints into. The following example creates a console and prints "Hello world!".

```
// Open a java console in DB-MAIN.
new DBMConsole();
// Print "Hello world!" in the console.
System.out.println("Hello world!");
```

3.2.3 DBMClassLoader

The class *DBMClassLoader* contains methods called by DB-MAIN to execute the java plug-ins. This class allows users to re-execute a modified java program without restarting DB-MAIN.

DB-MAIN does not use the system class loader because all classes loaded through it cannot be unloaded. Using an alternate class loader allows DB-Main users to edit source code of a class, recompile it and reload into DB-MAIN. To work, the loaded class must be in a different directory than jidbm.jar package.

Chapter 4

Programming styles

There are two ways of programming with the JIDBM library:

- The *classic style* that uses the JIDBM repository classes. The examples in sections 2.3.4, 5.1.2 and 5.2.2 use this style.
- The *library style* that directly uses the method of *DBMLibrary* class. The example below translates the classic example in section 2.3.4 into library style.

```
class Sample {
    public static void runDBM() throws IOException {
        // loading the project:
        DBMLibrary lib = new DBMLibrary();
        p = library.getObjectIdentifierOfGenericObject(0);
        if (p >= 0) {
            int s = lib.getFirstSchemaOfProject(p);
            if (s >= 0) {
                // analyzing the schema …
                s = lib.getNextSchemaOfProject(p,s);
            }
        }
    }
}
```

The library style programming can be more efficient than classic programming. In programs using the JIDBM repository classes, each class instantiation requires a few DB-MAIN repository accesses (to fill local and super-type fields) even when only one information is necessary. On big schemas, the response time must be slightly improved with the library programming.

The classic style is more ergonomic for programmers. It uses all the concepts (e.g. inheritance) which contributed to the success of the object programming.

Except for programs that requires innumerable repository accesses, we advice programmers to use the classic programming style that ensures efficient, readable and upgradable programs.

Chapter 5

Examples

5.1 Statistic generator

5.1.1 Description

The program displays some statistics on the current data schema:

- the number of entity types,
- the number of relationship types,
- the number of attributes,
- the maximal number of attributes by entity type.

5.1.2 Java code

```
import java.io.IOException;
import com.dbmain.jidbm.*;

public class StatisticGenerator {

    static int na_max;

    public static void runDBM() throws IOException {
        new DBMConsole();
        DBMLibrary lib = new DBMLibrary();
        if (lib.getCurrentSchema() >= 0) {
            DBMSchema sch = new DBMSchema(lib.getCurrentSchema());
            int ne = 0;
            int na  = 0;
            int nr = 0;
            na_max = 0;
            DBMDataObject d = sch.getFirstDataObject();
            while (d != null) {
                switch (d.getObjectType()) {
                case DBMGenericObject.ENTITY_TYPE:
                    ne = ne+1;
                    setAttributeMaximum((DBMEntityRelationshipType)d);
                    break;
                case DBMGenericObject.REL_TYPE:
                    nr = nr + 1;
                    break;
                case DBMGenericObject.SI_ATTRIBUTE:
```

```
                case DBMGenericObject.CO_ATTRIBUTE:
                    na = na + 1;
                    break;
                }
                d = sch.getNextDataObject(d);
            }
            System.out.println("\nSTATISTICS:"+"\n-----------\n"+
                               "#Entity types:\t"+ne+"\n#Rel-types:\t"+nr+
                               "\n#Attributes:\t"+na+
                               "\nMax attributes per entity: "+na_max+"\n");
        }
        else {
            System.out.println("\nNo selected schema!\n");
        }
        System.out.println("\nEnd...\n");
    }

    public static void setAttributeMaximum(DBMEntityRelationshipType e) {
        int max_at = 0;
        DBMAttribute a = e.getFirstAttribute();
        while (a != null) {
            max_at = max_at + 1;
            a = e.getNextAttribute(a);
        }
        if (max_at > na_max) {
            na_max  = max_at;
        }
    }
}
```

5.2 Schema creator

5.2.1 Description

The program creates a schema "Order" into a project loaded or created with DB-MAIN. This conceptual schema contains structures to manage customers, orders and products.

Note that the JIDBM library does not allow users to manage the structured histories of DB-MAIN. In this context, the created schema is only visible in the compact view of the project (menu View/Graph. compact). To create a schema in the project standard view, it must be created with DB-MAIN (menu New/Schema).

5.2.2 Java code

```
import java.io.IOException;
import java.util.*;
import java.text.*;
import com.dbmain.jidbm.*;

public class SchemaCreator {

    public static void runDBM() throws IOException {
        new DBMConsole();
        DBMLibrary lib = new DBMLibrary();
        DBMProject pro = new DBMProject();
        if (pro != null) {
            String sd = "20051214";
            SimpleDateFormat df = new SimpleDateFormat("yyyyMMdd");
            Date d;
            try {
                d = df.parse(sd);
            } catch (Exception ex) {
                d = null;
            }
            DBMSchema sch = pro.createSchema("Order","Ord","Conceptual",d,d,
```

```
                                          DBMSchema.ERASCHEMA,null);
        DBMUserView uv = sch.createUserView("",DBMUserView.GRAPH_STANDARD,
                                            (short)100,(short)100,0,0,
                                            (short)0,"",(short)0,"",
                                            (int)DBMGenericObject.MARK1,null);
        DBMEntityType cus = createCustomer(sch);
        DBMUserObject uo = uv.createUserObject(50000,50000,0,cus);
        DBMEntityType ord = createOrder(sch);
        uo = uv.createUserObject(100000,50000,0,ord);
        DBMEntityType prod = createProduct(sch);
        uo = uv.createUserObject(100000,100000,0,prod);
        DBMRelationshipType rel = sch.createRelationshipType("place","pla");
        DBMNote note = rel.createNote("A customer places orders\nto the "+
                                      "company.",sch);
        uo = uv.createUserObject(75000,20000,0,note);
        DBMRole rol = rel.createRole("",0,DBMRole.N_CARD,' ');
        rol.addFirstEntityType(cus);
        rol = rel.createRole("",1,1,' ');
        rol.addFirstEntityType(ord);
        rel = sch.createRelationshipType("detail","det");
        note = rel.createNote("A detail gives the ordered\nquantity "+
                              "of a product.",sch);
        uo = uv.createUserObject(150000,75000,0,note);
        rel.createSimpleAttribute("Quantity","Qu",1,1,' ',
                                  DBMSimpleAttribute.NUM_ATT,false,true,
                                  15,(short)2,sch,null);
        rol = rel.createRole("",0,DBMRole.N_CARD,' ');
        rol.addFirstEntityType(ord);
        rol = rel.createRole("",0,DBMRole.N_CARD,' ');
        rol.addFirstEntityType(prod);
    }
    else {
        System.out.println("\nNo loaded project!\n");
    }
    System.out.println("\nEnd...\n");
}

public static DBMEntityType createCustomer(DBMSchema sch) {
    DBMEntityType ent = sch.createEntityType("CUSTOMER","CUS");
    DBMSimpleAttribute id = ent.createSimpleAttribute("Ncus","Nc",1,1,' ',
                                                      DBMSimpleAttribute.NUM_ATT,
                                                       false,true,6,(short)0,sch,
                                                       null);
    DBMSimpleAttribute si = ent.createSimpleAttribute("Name","Na",1,1,' ',
                                                  DBMSimpleAttribute.VARCHAR_ATT,
                                                  false,true,30,(short)0,sch,id);
    DBMCompoundAttribute co = ent.createCompoundAttribute("Address","Add",1,1,
                                                           ' ',sch,null);
    ent.addNextAttribute(co,si);
    si = ent.createSimpleAttribute("Phone","Pho",1,5,DBMAttribute.SET_ATT,
                                   DBMSimpleAttribute.VARCHAR_ATT,false,true,15,
                                   (short)0,sch,null);
    ent.addNextAttribute(si,co);
    si = ent.createSimpleAttribute("Category","Cat",1,1,' ',
                                   DBMSimpleAttribute.VARCHAR_ATT,false,true,5,
                                   (short)0,sch,si);
    si = ent.createSimpleAttribute("Account","Acc",1,1,' ',
                                   DBMSimpleAttribute.VARCHAR_ATT,false,true,20,
                                   (short)0,sch,si);
    si = co.createSimpleAttribute("Street","Str",1,1,' ',
                                  DBMSimpleAttribute.VARCHAR_ATT,false,true,30,
                                  (short)0,sch,null);
    si = co.createSimpleAttribute("Zip-code","Zip",1,1,' ',
                                  DBMSimpleAttribute.NUM_ATT,false,true,10,
```

```
                                      (short)0,sch,si);
        si = co.createSimpleAttribute("City","Cit",1,1,' ',
                                      DBMSimpleAttribute.VARCHAR_ATT,false,true,30,
                                      (short)0,sch,si);
        DBMGroup gr = ent.createGroup("IDCUS",DBMGroup.ASS_GROUP,DBMGroup.PRIM_GR,0,
                                      1,null);
        gr.addFirstComponent(id);
        return ent;
    }

    public static DBMEntityType createOrder(DBMSchema sch) {
        DBMEntityType ent = sch.createEntityType("ORDER","");
        DBMSimpleAttribute id = ent.createSimpleAttribute("Nord","No",1,1,' ',
                                                          DBMSimpleAttribute.NUM_ATT,
                                                           false,true,6,(short)0,sch,
                                                           null);
        DBMSimpleAttribute si = ent.createSimpleAttribute("Date","Da",1,1,' ',
                                                         DBMSimpleAttribute.DATE_ATT,
                                                          false,true,15,(short)0,sch,
                                                           id);
        DBMGroup gr = ent.createGroup("IDORD",DBMGroup.ASS_GROUP,DBMGroup.PRIM_GR,0,
                                      1,null);
        gr.addFirstComponent(id);
        return ent;
    }

    public static DBMEntityType createProduct(DBMSchema sch) {
        DBMEntityType ent = sch.createEntityType("PRODUCT","");
        DBMSimpleAttribute id = ent.createSimpleAttribute("Npro","Np",1,1,' ',
                                                          DBMSimpleAttribute.NUM_ATT,
                                                           false,true,6,(short)0,sch,
                                                           null);
        DBMSimpleAttribute si = ent.createSimpleAttribute("Label","Lab",1,1,' ',
                                                      DBMSimpleAttribute.VARCHAR_ATT,
                                                      false,true,30,(short)0,sch,id);
        si = ent.createSimpleAttribute("Price","Pri",1,1,' ',
                                       DBMSimpleAttribute.NUM_ATT,false,true,15,
                                       (short)2,sch,si);
        si = ent.createSimpleAttribute("Stock","Sto",1,1,' ',
                                       DBMSimpleAttribute.NUM_ATT,false,true,15,
                                       (short)2,sch,si);
        DBMGroup gr = ent.createGroup("IDPRO",DBMGroup.ASS_GROUP,DBMGroup.PRIM_GR,0,
                                      1,null);
        gr.addFirstComponent(id);
        return ent;
    }

}
```