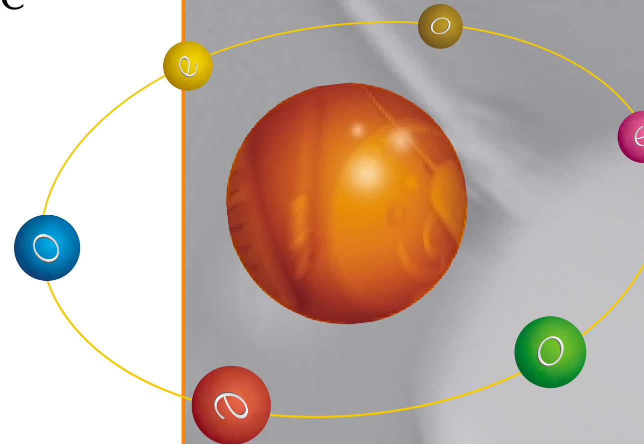
Objecteering/UML

Objecteering/SQL Designer User Guide

Version 5.2.2



Objecteering Software

www.objecteering.com

Taking object development one step further

Information in this document is subject to change without notice and does not represent a commitment on the part of Objecteering Software. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the purchaser's personal use, without the express written consent of Objecteering Software.

© 2003 Objecteering Software

Objecteering/UML version 5.2.2 - CODOBJ 001/002

Objecteering/UML is a registered trademark of Objecteering Software.

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

UML and OMG are registered trademarks of the Object Management Group. Rational ClearCase is a registered trademark of Rational Software. CM Synergy is a registered trademark of Telelogic. PVCS Version Manager is a registered trademark of Merant. Visual SourceSafe is a registered trademark of Microsoft. All other company or product names are trademarks or registered trademarks of their respective owners.

Contents

Chapter 1: Introduction
Overview of the Objecteering/SQL Designer module 1-3
Prerequisites 1-5
Developing applications 1-6
Structure of this user guide 1-8
Glossary 1-9

Chapter 2: Using the Objecteering/SQL Designer module
Working with the Objecteering/SQL Designer module 2-3
The properties editor for SQL 2-5

Chapter 3: First Steps
Overview of the first steps 3-3
Importing the demonstration package 3-4
Generating the physical model 3-6
Creating an SQL generation work product 3-9
Generating SQL files 3-11
Building the database schema: Executing the SQL 3-14

Chapter 4: Generation principles
Generating the database schema 4-3
Mapping generalization 4-5
Attributes 4-16
0..1-* associations 4-24
Generating triggers 4-28
Stored procedures 4-32
Compositions 4-34
1-1 associations 4-37
n-ary associations 4-38
Class associations 4-40
Generalization and associations 4-42
Structure in packages 4-46
Integrity constraints 4-47
Standardized logical schema 4-54
Additional annotations 4-55

Chapter 5: User interface
Interactive interface 5-3
The properties editor and the SQL Designer module 5-7
Generating the physical model 5-20
Generating the SQL 5-21
Visualizing the SQL 5-24
Executing the SQL 5-27
Executing the alter table file 5-29

Chapter 6: Annotating the model
Tagged value types 6-3
Note types 6-7
Stereotypes 6-9
Annotating an association 6-11
Annotating an association end 6-15
Annotating an attribute 6-17
Annotating a class 6-21
Annotating a component 6-29
Annotating a constraint 6-30
Annotating a datatype 6-32
Annotating a generalization 6-33
Annotating a model element 6-34
Annotating an operation 6-35
Annotating a package 6-39

Chapter 7: Module configuration
Overview of module configuration 7-3
Module parameter sets 7-4
Parameterizing by redefining J methods in a profile 7-14
Parameterizing the package unit 7-15
Parameterizing the class unit 7-16
Parameterizing the association unit 7-17
Parameterizing attribute units 7-19
Parameterizing by adding notes predefined by the generation 7-21

Chapter 8: Oracle Annex
Introduction 8-3
Class - Generation specific to Oracle 8-4
Class annotations 8-5
Association - Generation specific to Oracle 8-8
Association annotations 8-11
Attribute - Generation specific to Oracle 8-12
Attribute annotations 8-13
Distributed databases 8-14
Parameterizing generation - Specificity 8-16

Chapter 9: Sybase Annex
Introduction 9-3
Package annotations 9-5
Class - Generation specific to Sybase 9-6
Class annotations 9-7
Association - Generation specific to Sybase 9-10
Association annotations 9-14
Attribute - Generation specific to Sybase 9-16
Attribute annotations 9-17
Specific SQL errors 9-19
Parameterizing generation - Specificity 9-20
Restrictions 9-21

Chapter 10: SQL Server Annex
Introduction 10-3
Package annotations 10-4
Class - Generation specific to SQL Server 10-5
Class annotations 10-6
Association - Generation specific to SQL Server 10-7
Association annotations 10-8
Attribute - Generation specific to SQL Server 10-9
Attribute annotations 10-10
Specific SQL errors 10-11
Parameterizing generation - Specificity 10-12
Restrictions 10-13

Chapter 11: Calling on-line module commands
Calling on-line commands 11-3

Index

Chapter 1: Introduction

Overview of the Objecteering/SQL Designer module

Introduction

Welcome to the *Objecteering/SQL Designer* user guide!

The *Objecteering/SQL Designer* module is used to couple Objecteering/UML with RDBMS, and allows you to use a model to map the persistence of classes into a relational database.

Persistence characteristics are defined interactively through the simple annotation (tagged values) of different model units, whilst the generator creates tables, constraints and triggers.

Using the module

The modeling of an RDB application can be illustrated as shown in Figure 1-1 below.

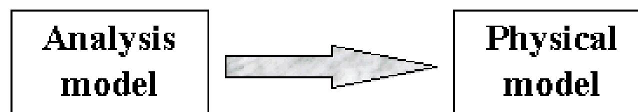


Figure 1-1. The two stages in the modeling of an RDB application

It is possible to pass directly from the analysis model (also called the logical model) to the physical model, by annotating the analysis model and applying transformation patterns to it, in order to generate the corresponding physical model.

The physical model obtained can also be annotated, so as to define specific SQL generation characteristics.

Operations are carried out using DDL (Data Definition Language) scripts. Here, SQL is used.

The RDBMS currently covered are Sybase, Oracle and SQL Server.

Specific features of the RDBMS target

Beyond the mapping provided as standard, the user can annotate his modeling using tagged values which allow the specification of production towards the relational, using the characteristics of the different RDBMS available on the market. For example, it is easy to group tables into clusters, or to add triggers and constraints.

Prerequisites

The Objecteering/UML environment

To work with the *Objecteering/SQL Designer* module, you must be familiar with the Objecteering/UML environment, detailed in the following user guides:

- *Objecteering/Administrating Objecteering Sites*
- *Objecteering/UML Modeler*

The *SQL Designer* module can only be used with the Enterprise and Professional Editions of Objecteering/UML.

Developing applications

Development steps

An application composed of persistent classes is developed by:

- designing a package
- generating the physical model
- creating tables (DDL)
- clarifying tables

Designing a package

The first step is the design of a package containing persistent classes. These classes are declared persistent by tagged values, which are used to specify the implementation of the persistence, by guiding the definition of the tables and keys used.

The "*SQL*" tab of the properties editor, described in the "*The properties editor for SQL*" section in chapter 5 this user guide, makes model annotation easier.

The tagged values provided by the *Objecteering/SQL Designer* module are described in chapter 6, "*Annotating the model*", of this user guide, as well as in the annex corresponding to the RDBMS used.

Generating the physical model

The *Objecteering/SQL Designer* module is used to generate and maintain one or several physical models from a logical model.

Generating SQL code

The user interface is used to:

- generate creation and destruction scripts for all the package's tables, views, stored procedures and triggers
- visualize the scripts produced
- run the scripts

The database is then created. The user can also use the free parts of SQL code to insert the first recordings into the database.

Adjusting the tables

This is not yet the end of the application's life. The generator is used to create individual tables which correspond to a class or an association. It remains possible to iteratively modify the logical model and to update the physical model(s) associated with it.

Structure of this user guide

The *Objecteering/SQL Designer* user guide is structured as follows:

- Chapter 2, *"Using the Objecteering/SQL Designer module"*: This chapter provides information on the module and how to use it.
- Chapter 3, *"First steps"*: A demonstration project will help you discover, step by step, the creation of tables in a database.
- Chapter 4, *"Generation principles"*: This chapter demonstrates the generation principles for the transformation of the logical model in Objecteering/UML into the relational model, by detailing the available mapping rules and options offered to the user. For each unit of the object structure model (class, attribute, generalization link, association), different possible mappings are detailed. Integrity checks which can be deduced from the model are presented at the end of the chapter.
- Chapter 5, *"User interface"*: This chapter offers a man/machine interface approach to the *Objecteering/SQL Designer* module. Starting with a package designed according to the principles developed in the previous chapter, the interface allows:
    - generation of the physical model from the logical model
    - interactive generation of the database diagram
    - visualization of the SQL produced and the execution of this SQL using the target RDBMS
- Chapter 6, *"Annotating the model"*: This chapter contains, for each model unit, all the "*tagged values*" common to all RDBMS handled by Objecteering/UML generation. This chapter represents a reference for users who have become familiar with the previous chapters.
- Chapter 7, *"Module configuration"*: This chapter details generation parameterization possibilities. The use of a type project and the configuration of generated scripts are also dealt with here.
- *Annexes*: for each RDBMS handled by the generator, an annex presents the required environment, data mapping, special cases of generation and specific tagged values.

Glossary

- *DDL (Data Definition Language)*: DDL code groups together SQL database schema creation and modification instructions (for example, create TABLE).
- *Generation work product*: (Objecteering/UML terminology) This indicates a source file generated by the CASE tool. A DDL generation work product is an SQL file.
- *Logical model*: This is an analysis model created by the user and annotated so as to allow the physical model to be deduced.
- *Physical model*: This is a relational model corresponding to a logical model. A physical model is made up of schemas, tables, views and utility classes containing stored procedures. These elements are represented by packages stereotyped <<schema>> and classes stereotyped <<table>>, <<sqlView>> and <<procedureClass>>.
- *Primary key*: The primary key of a relational table is composed of one or more attributes (the table's columns). It is used to clearly identify instances (in other words, an ordered list of values).
- *RDBMS*: Relational Database Management System. The RDBMS supported are Oracle, Sybase and SQL Server.
- *Type project*: This is a project which contains the mapping of Objecteering/UML types to the target RDBMS SQL types.

Chapter 2: Using the Objecteering/SQL Designer module

Working with the Objecteering/SQL Designer module

Installation

Before launching the tool, certain environment variables must be initialized.

Platform ...	Variable ...	Content ...	Role ...
PC, Unix	O_SQL_USER	UserName/Passwd/Server Name	allows you to connect to the DBMS to execute the generated DDL
PC, Unix	OBJING_PATH	path where Objecteering/UML is installed	used during installation

Module information

The *Objecteering/SQL Designer* module requires that Objecteering/UML be already installed (version 5.1.1 onwards). You must have the correct license in order to be able to use *Objecteering/SQL Designer*.

Module data is installed in the $OBJING_PATH/modules/SQLDesigner directory from the ObjecteeringModules database. This directory contains:

- doc
- externalization
- FirstStepsContainer
- res
- TypesPackageContainer

Creating a working environment for developing DDL scripts

To create a working environment used to develop DDL scripts, carry out the operations detailed below.

- *Create a new UML modeling project*. For details on how to create a UML modeling project, please refer to the "*Creating a new UML modeling project*" section in chapter 3 of the *Objecteering/Introduction* user guide.
- *Select the SQL Designer module for the new project*. For further information on how to select the module, please refer to the "*Selecting modules in the current UML modeling project*" section in chapter 3 of the *Objecteering/Introduction* user guide.

The properties editor for SQL

The properties editor is essentially a window designed to aid the user in his modeling, by providing rapid access to various information and services he may need to use.

The properties editor contains a number of tabs, including an "*SQL*" tab when the *Objecteering/SQL Designer* module has been selected for the current UML modeling project. This tab is used to:

- enter or modify certain information relevant to SQL generation on the element selected in the explorer, such as notes, tagged values or stereotypes
- generate SQL, visualize generated SQL and run SQL

For further general information on the properties editor, please refer to the "*The Properties editor*" section in chapter 3 of the *Objecteering/UML Modeler* user guide.

For information on the "*SQL*" tab of the properties editor, please refer to the "*The properties editor and the SQL Designer module*" section in chapter 5 of this user guide.

Chapter 3: First Steps

Overview of the first steps

Objective

This chapter will demonstrate how to build the persistent schema of a small library. It describes all the necessary steps used to obtain the package in the database.

The system to be modeled

Let us take an example to generate a relational database. In this example, a library system is modeled. This library stores books. The following example is for an Oracle database.

Importing the demonstration package

The "Library" package

We are now going to import the "*Library*" demonstration package. To do this, carry out the following steps:

1 - Create a new UML modeling project (in our example, the UML modeling project is called "*FirstStepsProject*"), and select the *SQL Designer* module (for further information on module selection, please refer to the "*Selecting modules in the current UML modeling project*" section in chapter 3 of the *Objecteering/Introduction* user guide).

2 - Right-click on the UML model root and run the "*SQL Designer/Import first steps*" command from the context menu which appears (as shown in Figure 3-1).

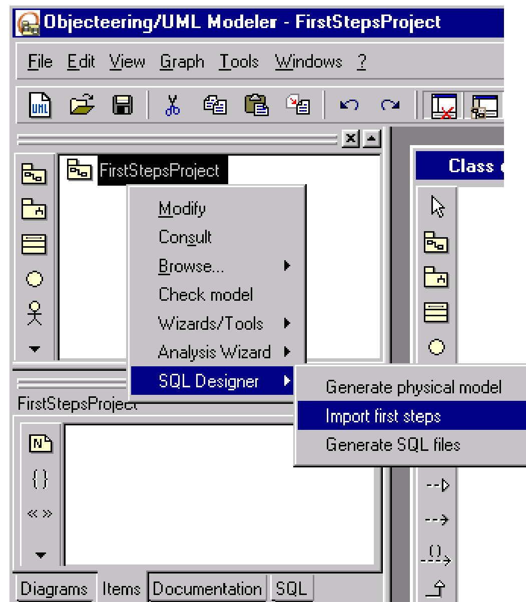


Figure 3-1. The "*Import first steps*" command

Class diagram

The class diagram shown in Figure 3-2 is available on the "*Library*" package.

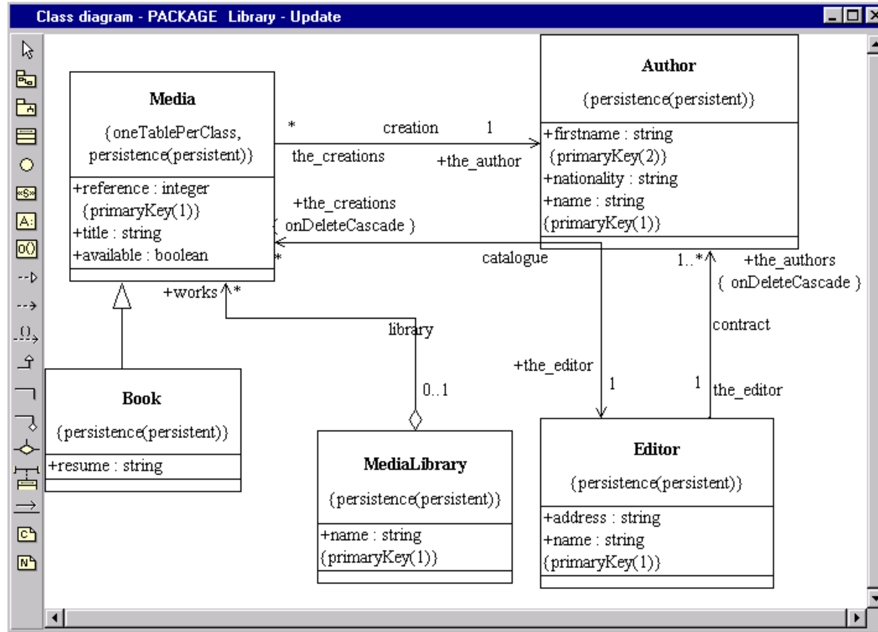


Figure 3-2. The "*Library*" package class diagram

Generating the physical model

To generate the physical model, carry out the steps shown in Figure 3-3.

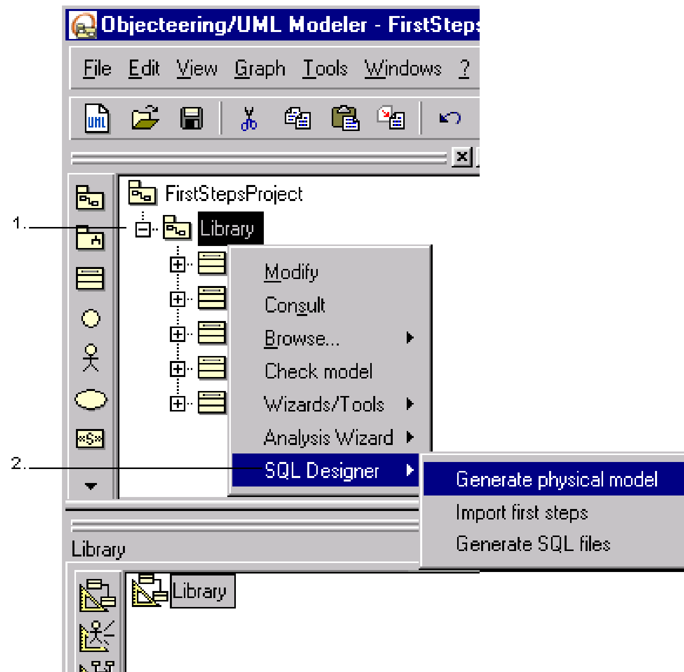


Figure 3-3. The "*Generate physical model*" command

Steps:

1 - Right-click on the "*Library*" package.

2 - Run the "*SQL Designer/Generate physical model*" command.

The same operation can be carried out in the "*SQL*" tab of the properties editor, as shown in Figure 3-4.

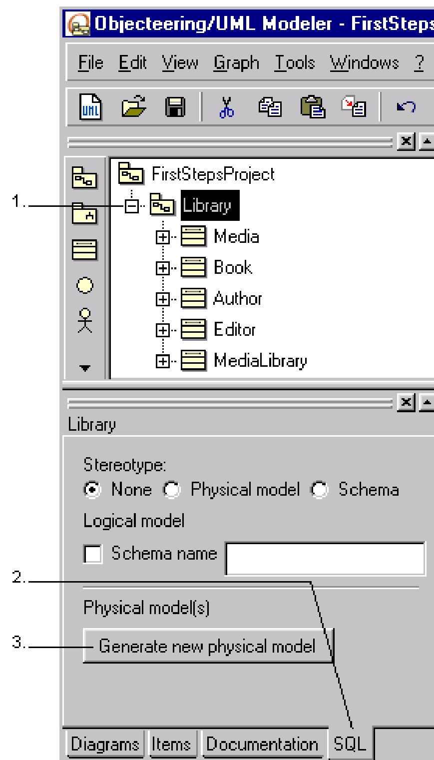


Figure 3-4. Creating the physical model in the "*SQL*" tab of the properties editor

Steps:

1 - Select the "*Library*" package in the Objecteering/UML explorer.

2 - Select the "*SQL*" tab in the properties editor.

3 - Click on the "*Generate new physical model*" button.

Whichever method is used, the physical model is then generated. A diagram is automatically generated (as shown in Figure 3-5).

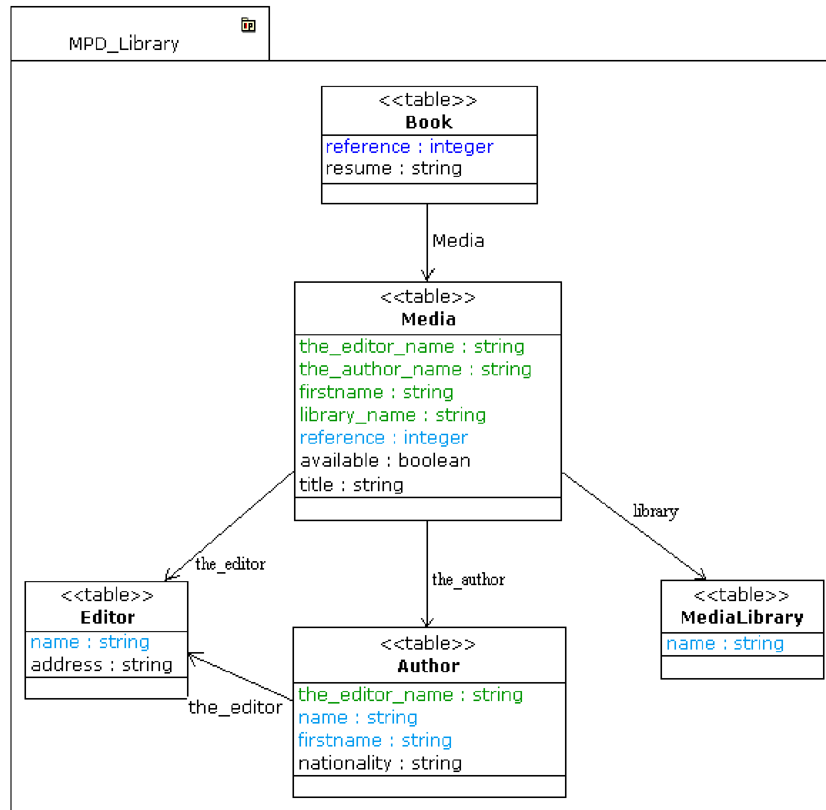


Figure 3-5. The diagram automatically generated for the physical model

Each class of the physical model represents a relational table, and is stereotyped <<table>>.

Creating an SQL generation work product

Procedure

We are now going to create an SQL generation work product for the physical model.

After following the procedure represented in Figure 3-6, we will then look at the dialog box for an SQL work product (Figure 3-7).

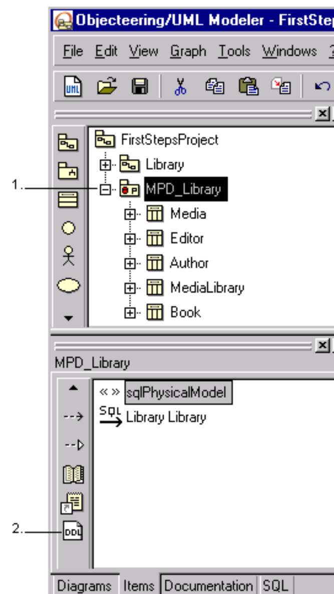
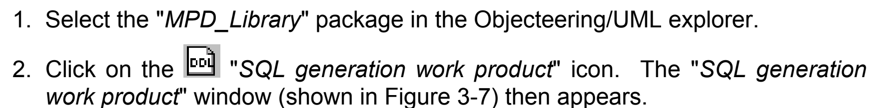


Figure 3-6. Creating a SQL generation work product for the "*MPD_Library*" package

Steps:

1. Select the "*MPD_Library*" package in the Objecteering/UML explorer.
2. Click on the  "*SQL generation work product*" icon. The "*SQL generation work product*" window (shown in Figure 3-7) then appears.

Defining a SQL generation work product

The SQL generation work product dialog box (shown in Figure 3-7) is used to indicate the generation directory for SQL scripts.

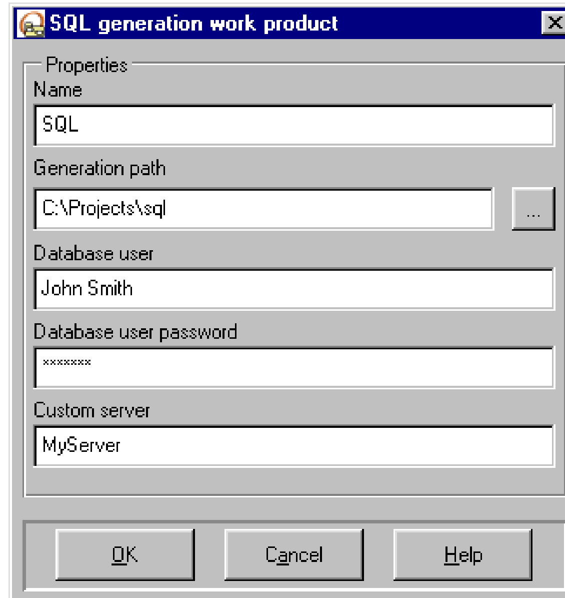


Figure 3-7. The "*SQL generation work product*" dialog box

Generating SQL files

Procedure

We are now able to generate the DDL (Data Definition Language), in other words, the SQL code which describes the database diagram, from the generation work product which we have just created, simply by following the procedure shown in Figure 3-8.

The Objecteering/UML console displays the generation steps and results.

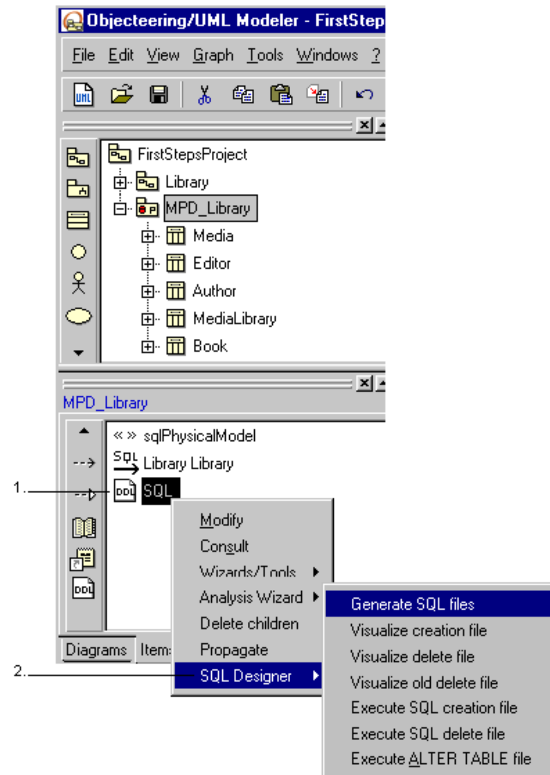


Figure 3-8. Generating the SQL files for the "*MPD_Library*" package

Steps:

1. Select the SQL generation work product in the "*Items*" tab of the properties editor using the right mouse-button.
2. Run the "*SQL Designer/Generate SQL files*" command from the context menu which appears.

Note: Double-clicking on the generation work product runs the "*Propagate*" command for this same work product.

The *O_SQL_USER* environment variable can be positioned, in order to allow access to the database from Objecteering/UML.

Example:

```
setenv O_SQL_USER User_name/Password/Host_name
```

Note: It is not necessary to position the *O_SQL_USER* environment variable in this way, as there exist module parameters for the user and his password, as well as parameters/attributes for the generation work product itself. However, this environment variable is necessary where a "*Host_name*" must be defined.

Visualizing the generated SQL

The SQL file is generated for the "*MPD_Library*" package and the classes that belong to it.

It is possible to visualize the generated SQL file from the Objecteering/UML explorer. The "*SQL Designer/Visualize creation file*" menu on the generation work product allows you to run the generated file's editor (as shown in Figure 3-9).

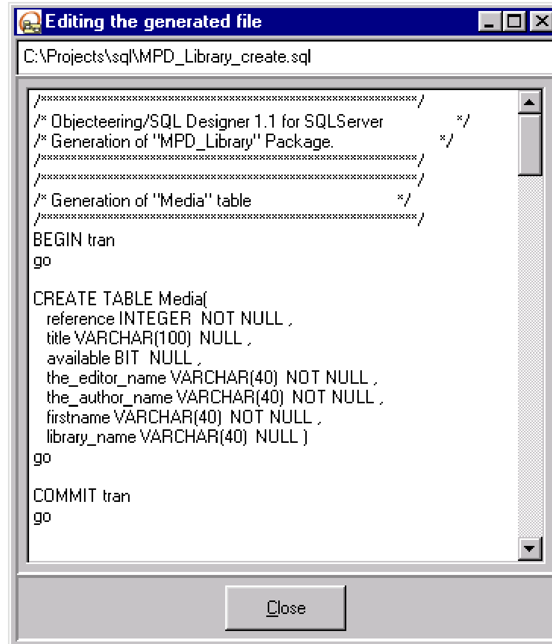


Figure 3-9. Editing the SQL creation file generated for the "*MPD_Library*" package

Note: The "*Visualize SQL*" button in the properties editor carries out the same operation as the "*Visualize creation file*" menu.

Building the database schema: Executing the SQL

Procedure

Taking the "*MPD_Library*" package as the starting point, we will now execute the SQL, by following the steps in Figure 3-10.

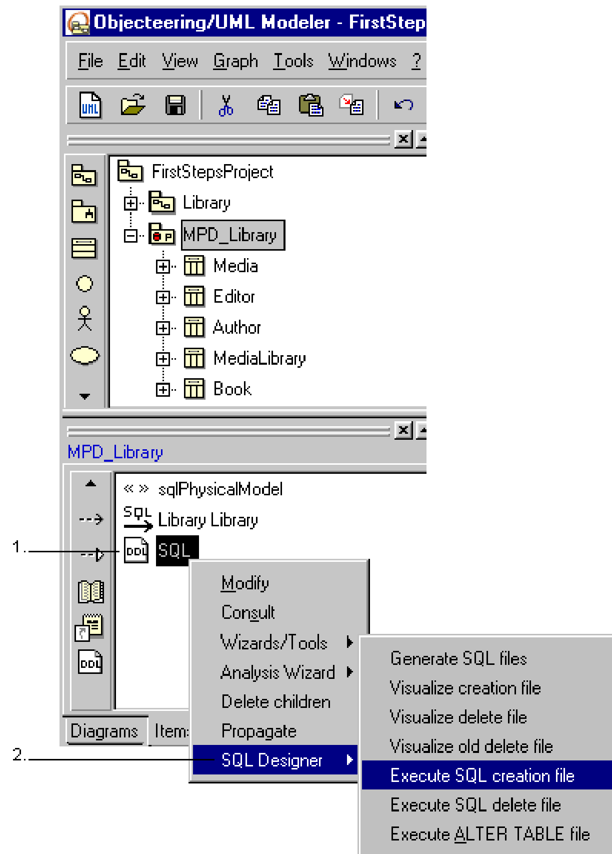


Figure 3-10. Executing the SQL for the "*MPD_Library*" package

Steps:

1. Select the SQL generation work product of the "*MPD_Library*" package with the right mouse-button.
2. Select the "*SQL Designer/Execute SQL creation file*" commands from the context menu which appears.

Chapter 4: Generation principles

Generating the database schema

General principle

Generation for relational databases takes into account the model in terms of its:

- classes
- associations
- generalizations
- attributes

Using tagged values

Tagged values describe specific RDBMS notions in the Objecteering/UML model. The characteristics of the target RDBMS (*cluster, structure*, etc.) can be accessed using additional tagged values.

Furthermore, parameterization allows the user to add his own tagged values, in order to his specific generation operations.

Mapping rules

The rules for mapping object model notions in the tables of a relational database are as follows:

The ... object model	relational databases ...
class	table
association	table or columns and outside keys
instance	not managed by the Objecteering/SQL Designer module
attribute	a table's column
generalization	3 alternatives: one table per concrete class (with copying specialized attributes) , one table per class(without copying specialized attributes), or only one table for the complete hierarchy
{primaryKey} tagged value	primary key
{index} tagged value	index
one or more packages	according to the stereotype (schema, database, physical model)

Example

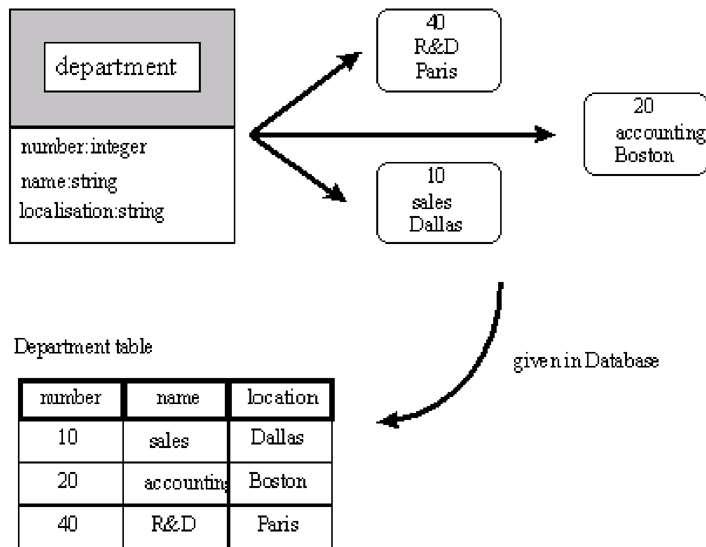


Figure 4-1. Example of mapping

This gives the following in SQL:

```
CREATE DEPARTMENT TABLE(
       NUMBER...,
       NAME...,
       LOCATION...);
```

Mapping generalization

Three mapping possibilities

Three principal techniques exist to solve the problem of table mapping:

1 - Copying the generalized attributes into all the tables which represent the child classes (the abstract classes of the persistent generalization tree are not then represented in any table).

2 - The basic class (which can be abstract) has its own database table. The tables of the child classes then reference it.

3 - All the classes of a generalization tree are mapped to a single table.

Choosing the right mapping technique

There is no ideal solution to the problem of selecting the mapping.

1 - *Solution 1* is normally quicker when running the program, since it does not necessitate any further links or complex operations in order to access the data. However, it requires larger tables, since it will be necessary to duplicate the definition of the generalized columns.

2 - *Solution 2* is less redundant, but there can be problems with the management of identifiers. If this is the case, it is necessary to manage unique primary keys for all the tables of the persistent generalization graph. Furthermore, access, like updates, will be more costly.

3 - *Solution 3* is the quickest when running the program, and allows the reading and writing of any BaseClass descendent with a single database operation. As all BaseClass descendents can be found in a single table, polymorphic reading is straightforward. The drawback is that the storing of object attributes requires more space than is absolutely necessary. This waste of space depends on the depth of the generalization tree. Furthermore, mapping too many classes to a single table may cause poor performance and database deadlock problems.

Example of mapping possibilities

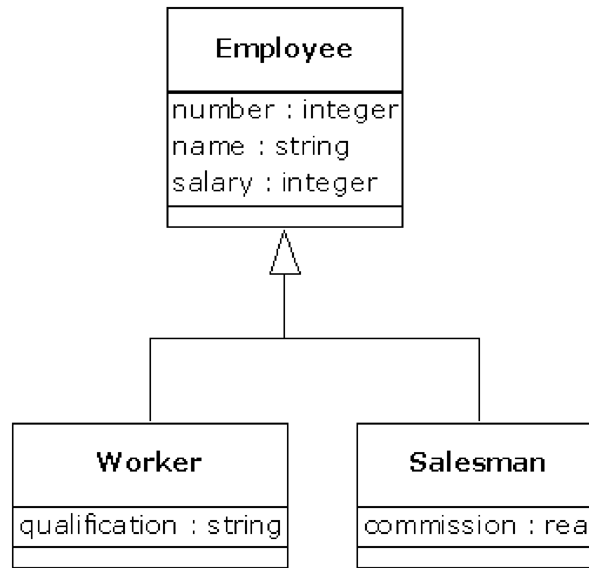


Figure 4-2. Example of mapping possibilities

Generalizing persistence

The persistence characteristic is a generalized characteristic. If the parent class is already persistent, generation detects that a child class is persistent. The *{persistence}* tagged value on derived classes is generally redundant (but nevertheless useful from a documentation point of view).

Example of generalizing persistence

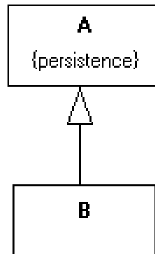


Figure 4-3. Example

Note: In this case, B is automatically persistent.

Selecting your mapping technique

The generalization mapping method is chosen at base class level, using the following tagged values:

1 - Solution 1: *{oneTablePerConcreteClass}*, which is used by default

2 - Solution 2: *{oneTablePerClass}*

3 - Solution 3: *{oneTable}*

Example with one table per concrete class

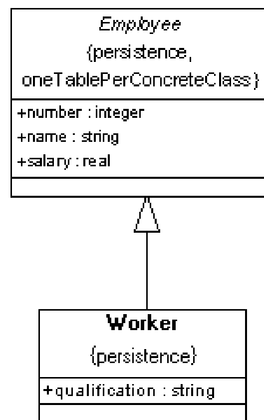


Figure 4-4. Example of mapping with the *{oneTablePerConcreteClass}* tagged value

Note: The tagged value on the "*worker*" class serves no purpose.

This maps the worker class as follows:

```
CREATE WORKER TABLE   {
  NUMBER ...,
  NAME ...,
  SALARY ...,
  QUALIFICATION ...
};
```

The result is shown in Figure 4-5 below.

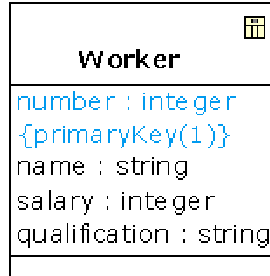


Figure 4-5. Result of mapping with the *{oneTablePerConcreteClass}* tagged value

Generalization of the generation mode

If the mode for generating persistence is not defined in the current class, and owns a parent class, the mode will be generalized by it. An error is detected if the generalization type defined locally is incompatible with the previous rule.

Example with one table per class

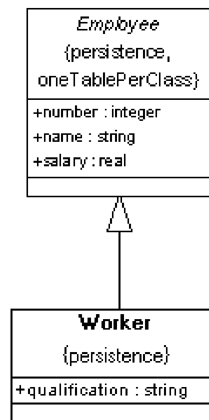


Figure 4-6. Example of mapping with the *{oneTablePerClass}* tagged value

This maps the worker class as follows:

```
CREATE WORKER TABLE
   NUMBER ...,
   QUALIFICATION ...,
} ;
```

The result is shown in Figure 4-7 below.

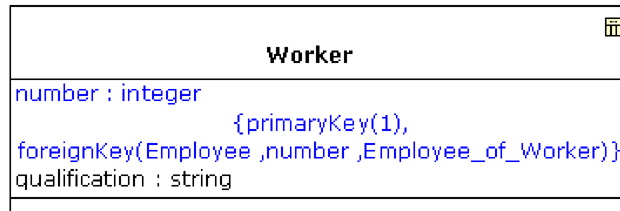


Figure 4-7. Result of mapping with the *{oneTablePerClass}* tagged value

Example with one table

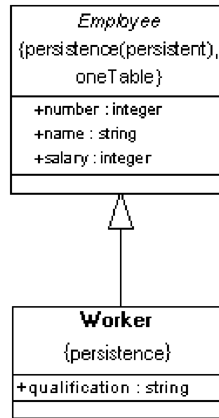


Figure 4-8. Example of mapping with the *{oneTable}* tagged value

This maps both the employee and the worker classes as follows:

```
CREATE EXPLOYEE TABLE
   NUMBER ...,
   NAME ...,
   SALARY ...,
   QUALIFICATION ...,
} ;
```

The result in shown in Figure 4-9 below.

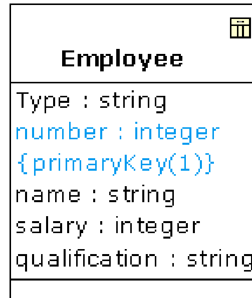


Figure 4-9. Result of mapping with the *{oneTable}* tagged value

Without specifying the generalization mapping mode, the "*one table per concrete class*" mode is adopted.

If *{oneTablePerClass}* had been defined at the level of the "*Worker*" class, there would have been an error!

Applying this rule allows the user to mix three mapping modes within the same database diagram, while fixing only one type of persistency per simple generalization tree.

Multiple generalization

This version of the module does not take into account the generalization of several persistent classes. Using persistent multiple generalization provokes an error during the generation of the package.

However, it is possible that a persistent class specializes a persistent class and one or more non-persistent classes.

Abstract class

The *Objecteering/SQL Designer* module does not produce any tables corresponding to a derived abstract class with the "*one table per concrete class*" pattern. Indeed, such tables cannot contain any recordings.

Integrity

The current version does not generate any uniqueness constraints on keys between several classes which specialize the same class with the "*one table per concrete class*" pattern.

Generalization with the "*one table per class*" pattern implies the creation of outside keys in the child classes, which allow you to guarantee the existence of a recording in a parent class.

The automatic deletion by reflex of recordings through references from child classes towards their parent classes ("*delete on cascade*" notion) is not carried out.

Attributes

"Atomic" attributes

The mapping of attributes "*in columns*" implies that only atomic attributes should be used: we thus obtain the first normal form. Therefore, basic types (or predefined primitive classes) must have a relational non-decomposable equivalent.

Type mapping

The action of mapping attributes poses a problem commonly known as the "*impedance problem*" (because of the similarity to problems in electricity). An object-oriented information structure has to be transformed in a table-oriented structure, in other words, language types have to be converted into RDBMS primitive types.

Generation suggests, by default, a global conversion policy that can be adapted and increased by the user (see chapter 7, "*Module configuration*", of this user guide for further details). The table below presents the mapping rules for the Oracle RDBMS.

Model	**ORACLE Data Types**
integer	INTEGER
{short} integer	INTEGER
{long} integer	INTEGER
{unsigned} integer	INTEGER
boolean	INTEGER
real	FLOAT
{long} real	FLOAT
string(n)	VARCHAR2(n)
char	CHAR (1)
set {array} (n) of char	VARCHAR2 (n)

Note: The generator does not check the uniqueness of the names in the SQL sense, not being case sensitive. This type of error, therefore, only appears when the SQL script is executed.

Access restriction

Attributes can be declared public, protected or private.

Class attributes

A class attribute is unique for the set of instances of the class. In the context of databases, this concept should be clarified as follows: a class attribute is unique for the database which memorizes it.

Two transformations are possible:

- By the creation of a table which groups together all the class attributes of all the application's classes
- By the creation of a specific table associated with the class on which one or several class attributes are defined.

The first transformation is simple to implement, and limits the number of tables. All class attributes for all classes are grouped in a table, indicated by the parameter of the {*globalTable*} tagged value on the class attributes. Each line of this table contains the value of a class attribute.

However, this pattern presents a major disadvantage, in as much as it requires the definition of one column per type of class attribute. For example, if five class attributes are present, three of which are integer type and two string type, the global table will contain two columns:

- an "*att_int*" integer-type column
- an "*att_string*" string-type column

The table structure also includes a "*ref_attribute*" attribute, to contain the name of the class and of the class attribute. It is possible to have several tables which group together class attributes.

The following model (as shown in Figure 4-10) presents two class attributes which will be grouped in a global table.

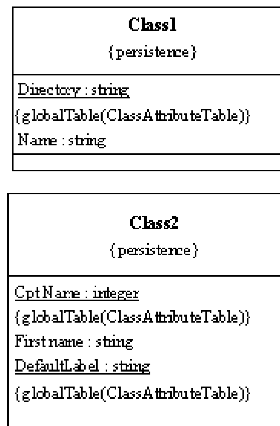


Figure 4-10. The analysis model, modeling two class attributes

Figure 4-11 shows the physical model after transformation and automatic generation of a global table.

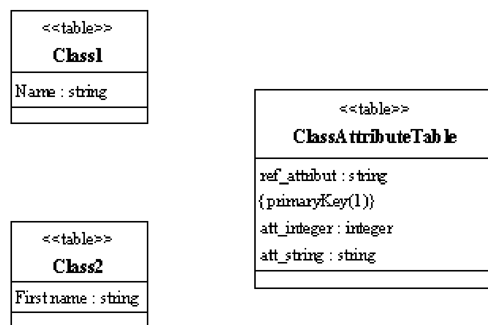


Figure 4-11. The physical model, implementing a global table for class attributes

Figure 4-12 presents the "*ClassAttributeTable*" table for the "*Class1*" and *Class2*" classes.

Table1 : Table

ref_attribut	att_integer	att_string
Class1_Directory	0	c:\tmp
Class2_CptName	164	
Class2_DefaultLabel	0	Label to add

Figure 4-12. Example of a table containing class attributes

The second transformation consists of the definition of one table per class, including one or several class attributes. This transformation is more in line with the class attribute notion, despite the generation of a specific table. No annotation is necessary, and a finer management of attributes and their associated type is possible. The generated table bears the *"classname_attribute*" name. This name can be modified, by using the *{classAttributesTableName}* tagged value.

Figure 4-13 shows the analysis model.

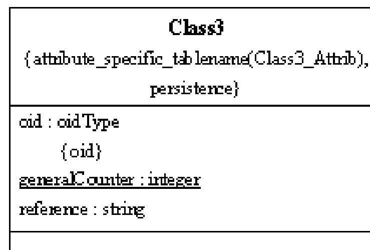


Figure 4-13. Analysis model, modeling a class attribute for the generation of a specific table

Figure 4-14 shows the corresponding physical model.

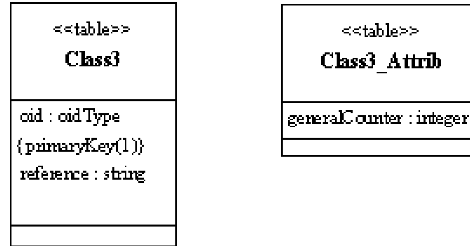


Figure 4-14. Physical model, generating a class attribute in a specific table

The {sqlDefault} value

In Oracle or Sybase, the *DEFAULT* clause is used to specify the value which will be assigned to a column if the insertion of a set of values misses a value for this column. In SQL coupling, this feature is implemented using the {*sqlDefault*} tagged value on attributes.

Note: This notion is separate from the notion of initial value in the model (taken into account in dynamic code). Specifying this value in an attribute's dialog box has no effect whatsoever on SQL generation. The properties editor can be used to enter this tagged value.

The {sqlType} tagged value

This tagged value is used to map specific types. For further information, please refer to:

- the "*Parameterizing generation - Specificity*" section in chapter 8 of this user guide for Oracle
- the "*Parameterizing generation - Specificity*" section in chapter 9 of this user guide for Sybase
- the "*Parameterizing generation - Specificity*" section in chapter 10 of this user guide for SQL Server

Primary key definition

The primary key of a relational table is composed of one or more attributes (the table's columns). It allows the clear identification of the instances (in other words, the sets of values).

Second normal form: the primary key must be composed of the smallest combination of attributes which are used in identification.

The {primaryKey} tagged value

The user must define the primary key of the associated table for a persistent basic class using the *{primaryKey(rank)} tagged value*, with as parameter the rank of the attribute in the primary key. Its declaration consequently forbids null values for columns thus specified, as well as guaranteeing the key's uniqueness. This uniqueness constraint will be mentioned in the following paragraphs. Just like {*persistence*}, this tagged value is generalized.

Note: Key order can influence access to the database. It is preferable to rank its components from the most discriminating to the least selective. A button in the properties editor on a class and a menu on classes can be used to open a window used to graphically define the primary key.

Example

gradesal
{persistence(persistent)}

grade : string
{primaryKey(1)}
level : integer
{primaryKey(2)}
salmin : real
salmax : real

Figure 4-15. Defining the primary key

In other words, in SQL (without taking into account the key's uniqueness constraint):

```
CREATE TABLE GRADESAL (
      GRADE... NOT NULL,
      LEVEL... NOT NULL,
      MINSAL...,
      MAXSAL...
);
```

Uniqueness of the primary key

The declaration of a primary key brings about an integrity constraint. Each time sets of values are created, it is necessary to check that the value of the key does not already exist in the database, in order to preserve the uniqueness of the primary key.

As the primary key is also a priority access key to the information, this constraint is implanted via the declaration of a "unique index", the generation of which depends on the target RDBMS. For example, Oracle uses the "PRIMARY KEY" command. The name of this index is the name of the table with the "_PK" suffix.

This gives the following in SQL:

```
CREATE TABLE GRADESAL (
      GRADE... NOT NULL,
      LEVEL... NOT NULL,
      MINSAL...,
      MAXSAL...
);
CREATE UNIQUE INDEX GRADESAL_PK
      ON GRADESAL (GRADE, LEVEL);
```

Generalization and the primary key

The primary key of a child class is obligatorily the same as the parent class' primary key.

0..1-* associations

Patterns for transforming 0..1-* associations

A 0..1-* association defined between two classes is mapped by implementing one or several class attributes on the * side of the association. Attributes which define foreign keys are automatically generated and must correspond to the primary key defined on the class on the 0..1 side of the relationship.

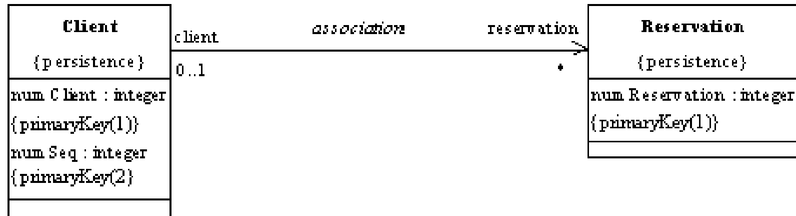


Figure 4-16. Analysis model, modeling 0..1-* relationships

Figure 4-17 shows the above model after transformation.

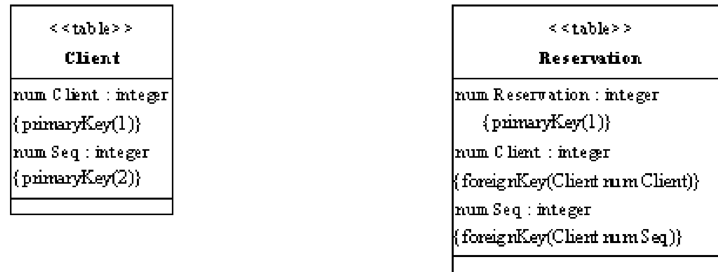


Figure 4-17. Physical model, modeling a 0..1-* association

Classes annotated *{persistence(persistent)}* have been transformed into classes stereotyped *<<table>>*. Attributes annotated *{primaryKey}* will allow, for each table, the generation of constraints linked to primary keys.

Primary keys defined on the "*Client*" class have been taken into account in the "*Reservation*" class, in the form of foreign keys. The names of the attributes defined as foreign keys are generated as follows: *"role_attributename"*.

Note: To simplify the diagrams of this specification, the names of the roles have voluntarily been omitted.

By default, role names are not used in model transformation. If a naming conflict is detected on the generated attributes (several attributes with the same name having to be generated in one table), the name of the association's role is added before the name of the attribute.

It should be noted that the columns created can be defined as being NULL if the minimum multiplicity is zero. A constraint stereotyped <<notNull>> is added to the generated attributes if the multiplicity is 1 on the aggregation side. This constraint is used to add a NOT NULL clause to SQL file generation.

The *{foreignKey}* tagged value is used to reference the corresponding attributes, in order to allow the generation of the correct integrity constraints. This transformation of the 0..1 multiplicity of the association does not generate intermediary tables, whatever the browsing direction of the association.

Associations of aggregation type (empty diamond), which model weak coupling between instances, are considered as associations, as far as the transformation of the analysis model into a physical model is concerned.

The model illustrated in Figure 4-18 presents an aggregation between two classes. The transformation applied to this type of relationship is identical to that of an association (see Figure 4-17, "*Physical model, modeling a 0..1-* association*").

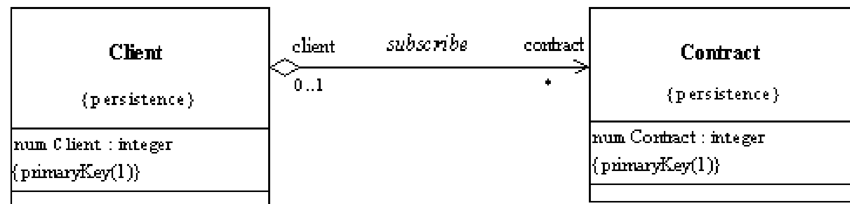


Figure 4-18. Analysis model, modeling a 0..1-* aggregation

For associations whose multiplicity is 1-*, attributes which define foreign keys are generated with the *{not_null}* tagged value, so as to respect modeled multiplicity. An example is shown in Figure 4-19.

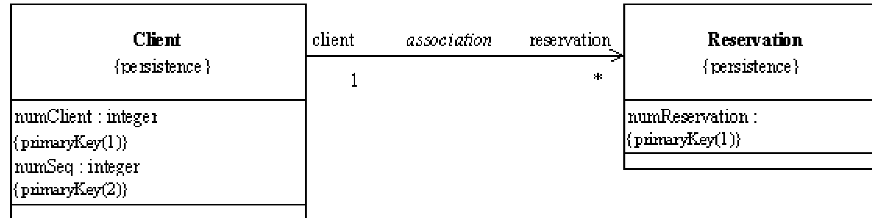


Figure 4-19. Analysis model, modeling a 1-* association

Complex modeling can cause the appearance of scenarios, for which the transformation rules presented below cannot be applied. It is then necessary to specify the associated SQL elements for each relationship between classes (associations, compositions, aggregations).

The following analysis model (Figure 4-20) for which the association is annotated *{external}* forces the generation of an intermediary table, to map the association. Here, the *{sqlName}* tagged value is used to specify the name used to name the generated intermediary table. If no tagged value is present, a name is automatically generated from the name of the end tables and the name of the association.

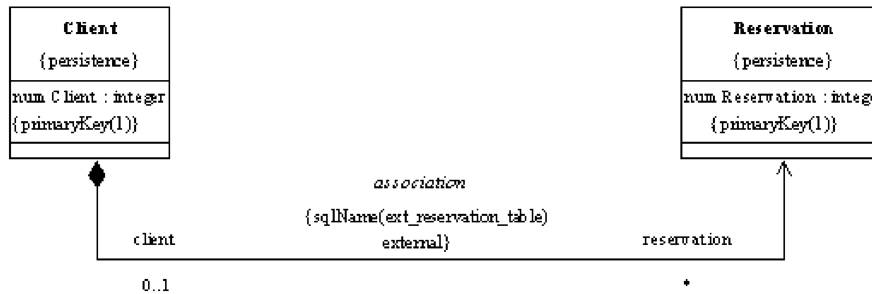


Figure 4-20. Analysis model, modeling a 0..1-* composition with an intermediary table

The analysis model shown in Figure 4-20 is transformed as shown in Figure 4-21.

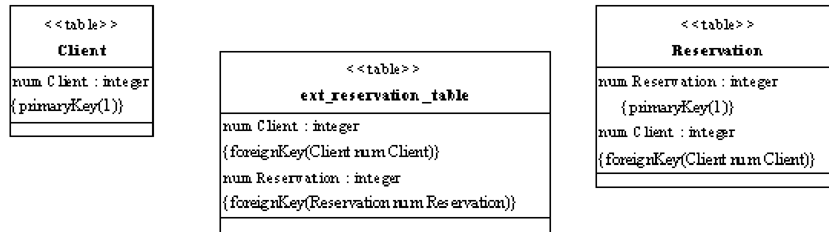


Figure 4-21. Physical model, modeling a 0..1-* composition with an intermediary table

According to the transformation pattern applied, naming conflicts for attributes can appear. It is possible to give two attributes in two different classes the same name. In this case, the name of the role is used as discriminator. The use of the *{sqlName} tagged value* defined on an attribute is another way of getting round this problem, but can lead to the modification of the attribute name in all generations in which it takes part.

Cyclic associations

It is possible to specify cyclic associations (in the RDBMS sense). This does not present any particular problem for the generator. However, it is important to note that the generation of tables can only be carried out through the SQL script of the entire package. Indeed, as the outside keys refer to tables that have not yet been created, so the SQL files that correspond to the classes use each other.

As the method forbids mutual use between packages, the problem cannot arise between two of the model's packages.

Managing instances and "all" multiplicity

From a structure UML profile, associations which implement the management of persistent instances, as well as those whose multiplicity is "*all*" (the latter assume that the destination class has itself instance management) do not require any particular mapping. Indeed, they are already directly represented in a database using the tables associated to the persistent destination classes.

Generating triggers

Multiplicity constraints

Objecteering/SQL Designer uses triggers to map multiplicity constraints in associations, for Oracle, Sybase and SQL Designer. Multiplicity constraints are checked by a trigger before and after insertion for Sybase and SQL Server, and after insertion for Oracle.

For further information on using triggers to map multiplicity constraints, please refer to the relevant annex at the end of this user guide.

Figures 4-22 and 4-23 show an example of the use of triggers to map multiplicity constraints, Figure 4-22 showing the starting model and Figure 4-23 the physical model.

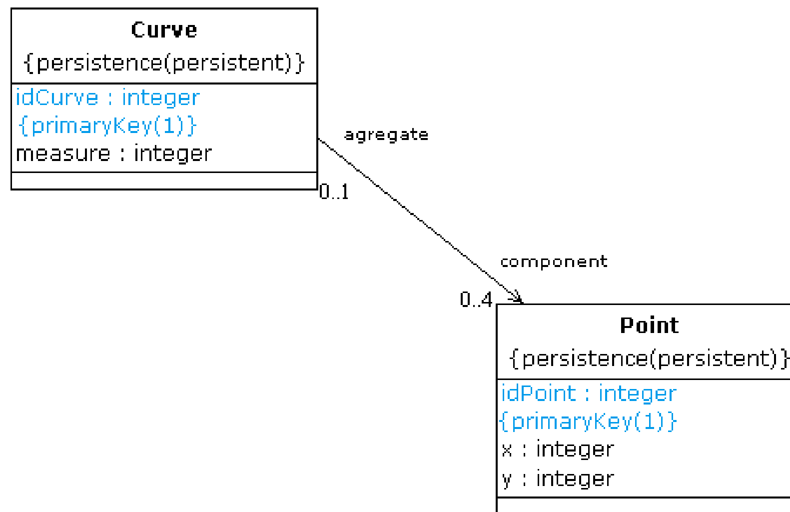


Figure 4-22. The starting model

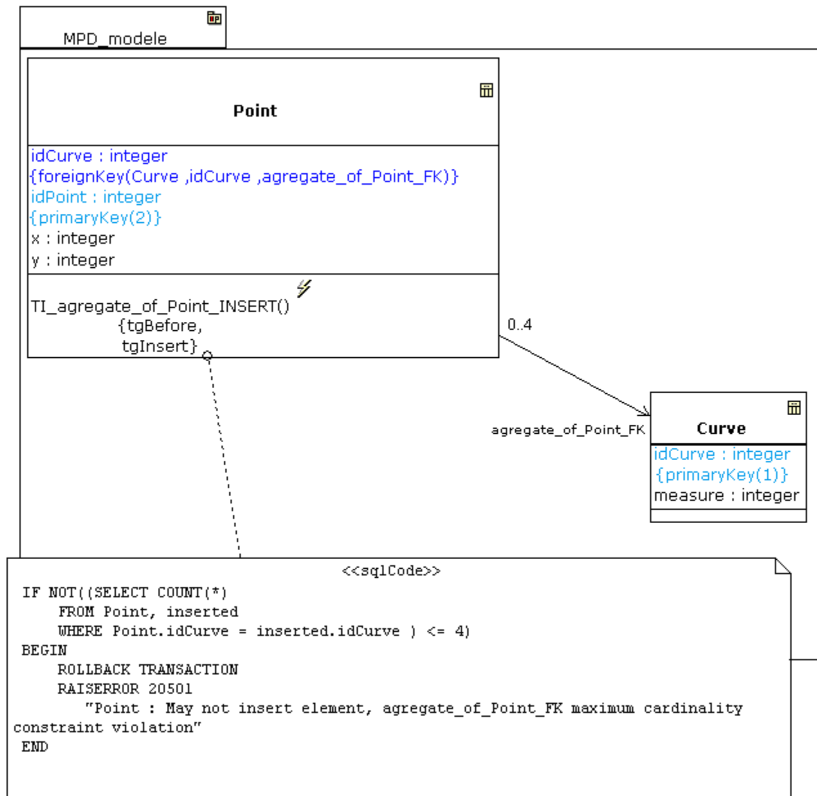


Figure 4-23. The physical model

The generated SQL file is as follows:

```
/*****************************************************/
/* Objecteering/SQL Designer 1.1.g for SQLServer      */
/* Generation of "MPD_modele" Package.                */
/*****************************************************/
/*****************************************************/
/* Generation of "Curve" table                        */
/*****************************************************/
BEGIN tran
go

CREATE TABLE Curve(
   idCurve INTEGER  NOT NULL ,
   measure INTEGER  NULL )
go

COMMIT tran
go


/******************************************************/
/* Generation of "Point" table                         */
/******************************************************/
BEGIN tran
go

CREATE TABLE Point(
   idCurve INTEGER  NULL ,
   idPoint INTEGER  NOT NULL ,
   x INTEGER  NULL ,
   y INTEGER  NULL )
go

COMMIT tran
go


/*******************************************************/
/* Generation of <MPD_modele> Package table primary key  */
/* constraints                                          */
/*******************************************************/
BEGIN tran
go
```

```
ALTER TABLE Curve ADD
   CONSTRAINT Curve_PK PRIMARY KEY (idCurve)

go

ALTER TABLE Point ADD
   CONSTRAINT Point_PK PRIMARY KEY (idPoint)

go

COMMIT tran
go


/*********************************************************/
/* Generation of "TI_agregate_of_Point_INSERT" trigger on
*/
/* "Point" table                                         */
/*********************************************************/
BEGIN tran
go

CREATE TRIGGER TI_agregate_of_Point_INSERT
 ON Point
 FOR INSERT
 AS
 IF NOT((SELECT COUNT(*)
     FROM Point, inserted
     WHERE Point.idCurve = inserted.idCurve ) <= 4)
 BEGIN
     ROLLBACK TRANSACTION
     RAISERROR 20501
        "Point : May not insert element,
agregate_of_Point_FK maximum
cardinality constraint violation"
 END


go

COMMIT tran
go
```

Stored procedures

Overview

Stored procedures are operations annotated <<storedProcedure>>, which can be modeled in the logical or the physical model.

Figure 4-24 and the code which follows show an example of a stored procedure.

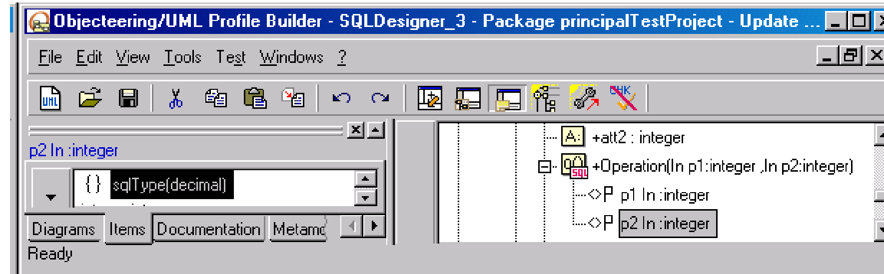


Figure 4-24. Example of a stored procedure

The generated SQL file will contain the following for SQL Server:

```
/***************************************************/
/* Generation of "Operation" stored procedure      */
/***************************************************/
BEGIN tran
go

CREATE PROCEDURE Operation (
   @p1  INTEGER,
   @p2  DECIMAL )
 AS
..
..


go

COMMIT tran
go
```

Stored procedure parameter types

As with attributes, the types of stored procedure parameters can be customized using either the properties editor or the {sqlType} tagged value. For further information on type mapping, please see the "*Attributes*" section in the current chapter of this user guide.

For further information on the {sqlType} tagged value, please refer to:

- the "*Parameterizing generation - Specificity*" section in chapter 8 of this user guide for Oracle
- the "*Parameterizing generation - Specificity*" section in chapter 9 of this user guide for Sybase
- the "*Parameterizing generation - Specificity*" section in chapter 10 of this user guide for SQL Server

Compositions

Model and relational

The mapping of compositions in a relational model is optimized with regard to the general case of an association. This specific mapping allows you to save space in the database and to improve performance when loading tables into the memory.

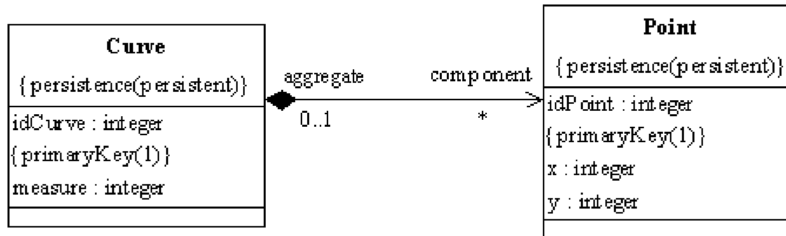


Figure 4-25. Example of a composition

Generated tables

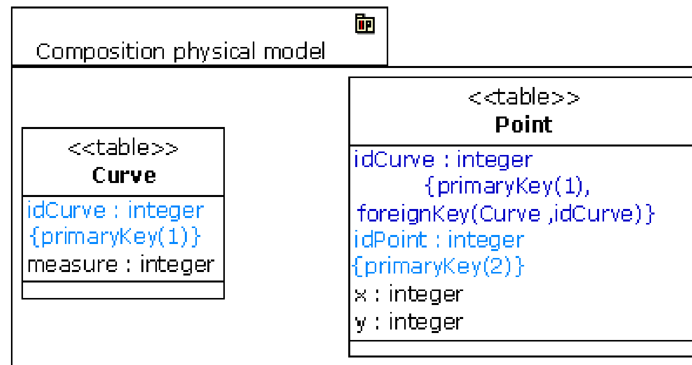


Figure 4-26. Example of a composition

```
Curve  (idCurve, measure)
Point  (idCurve, idPoint, abscissa, co-ordinates)
```

The association is materialized by copying the primary key(s) of the aggregate class, here "*Curve*", into the table of the class which has the role of component, here the "*Point*" class.

"*Component*" type objects are always handled through the "*aggregate*" class. The addition, removal and update of objects which are components of a composition is carried out by modifying "*aggregate*" objects in the memory, and by asking for an update of this object in the database.

Application domain

This optimized mapping of a composition, a special instance of an association, cannot be applied in the cases detailed below:

- The "*component*" class of the composition cannot belong to a generalization graph except as a root class.
- The "*component*" class cannot be component of more than one class.

When one of the clauses is not respected, the mapping of general associations will be adopted.

1-1 associations

Pattern for taking into account associations of 1-1 type

Associations of 1-1 multiplicity can be transformed in two different ways:

- By merging the two tables
- By generating a foreign key on both sides

The first method limits the number of tables. It consists of merging the two tables into one. The primary keys of each of the modeled classes are assembled to create the primary key of the resulting table. After model transformation, it is possible to delete the definition of a *{primaryKey}* tagged value, in order to limit the attributes used as the primary key of the table obtained. In this case, subsequent transformation of the analysis model into the physical model will generate the attribute with an associated *{primaryKey}* tagged value. Attributes should be annotated {*sqlDoNotUpdate*}. This transformation is effective if a role has the *{sqlOptimized} tagged value*. The destination table is then merged in the source table.

Depending on the model of the application, this first solution may not suit, because of the association(s) from these classes towards or from other classes.

The second method allows the generation of two distinct tables, and strictly respects the analysis model, by transforming each of the classes into a table stereotyped <<table>>. In this case, attributes which define the primary keys of each of the two tables are generated in the other table as foreign keys. This transformation is applied by default.

n-ary associations

n-ary association pattern

The transformation of an n-ary association is mapped through the generation of an intermediary table stereotyped <<*table*>>, which contains the set of primary keys for the different classes positioned at the ends of the association.

The following example (Figure 4-27) shows the transformation of an association between three persistent classes.

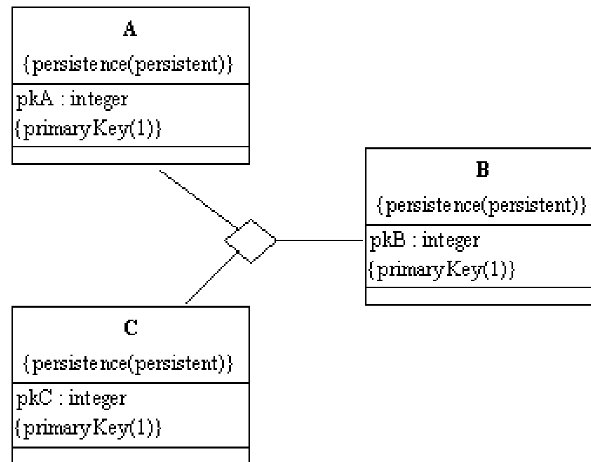


Figure 4-27. Analysis model, modeling an n-ary association

After transformation, an intermediary table has been generated. This table is made up of the set of attributes modeled as primary key of the classes which take part in the n-ary association.

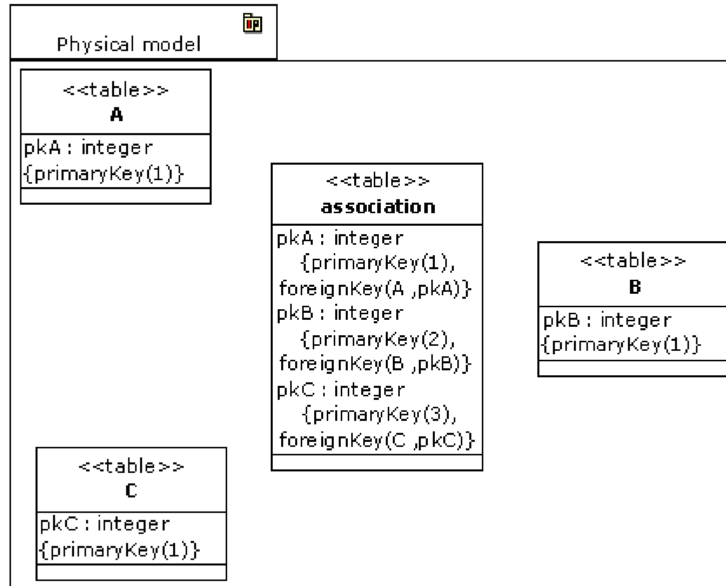


Figure 4-28. Physical model, modeling an n-ary association

- association pattern

An association of *-* multiplicity is treated as an n-ary association. For further information, please see the previous paragraphs.

Note: An association which is annotated {*external*} is managed like a *-* association.

Class associations

Class association patterns

The transformation of an association including a class association depends on its multiplicity and on the presence of the *{persistence(persistent)}* tagged value.

Default transformation consists of generating an intermediary table in the same way as a *-* association. The class association is used as intermediary table (please refer to the "*n-ary associations*" section of the current chapter of this user guide for further information).

For an association of 0..1-* multiplicity, and in the aim of limiting the number of tables, the class association can be generated in the table bearing the "*" multiplicity, if the role is annotated *{sqlOptimized}*. In this case, all the attributes of the class association are transferred (except those annotated *{persistence(transient)}*).

The analysis model shown in Figure 4-29 presents this possibility.

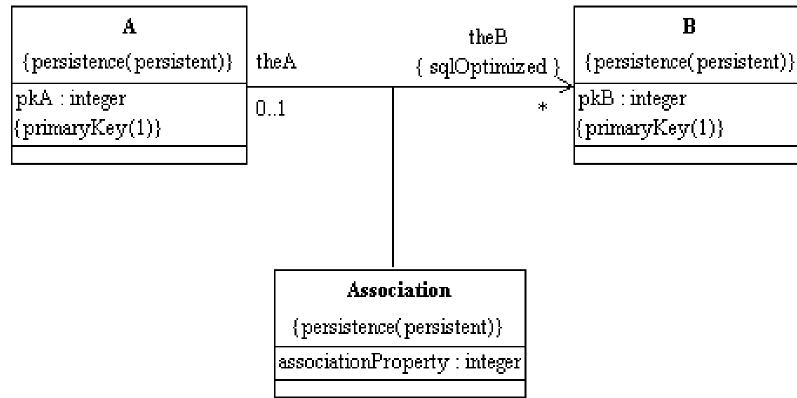


Figure 4-29. Analysis model, optimizing a class association

It is generated as shown in Figure 4-30.

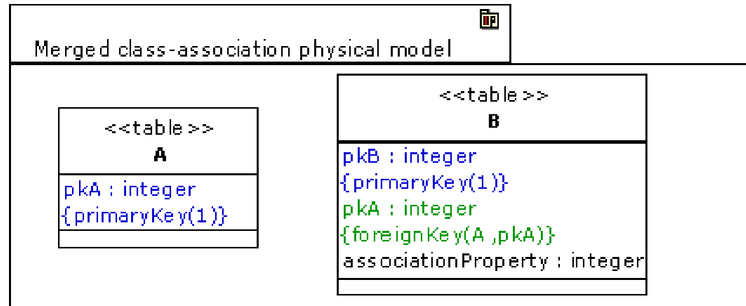


Figure 4-30. Physical model, optimizing a class association

Generalization and associations

Generalization and associations

From the UML profile of a table, associations are not specialized as such. The associated table contains all the references, including the child class instances.

Example:

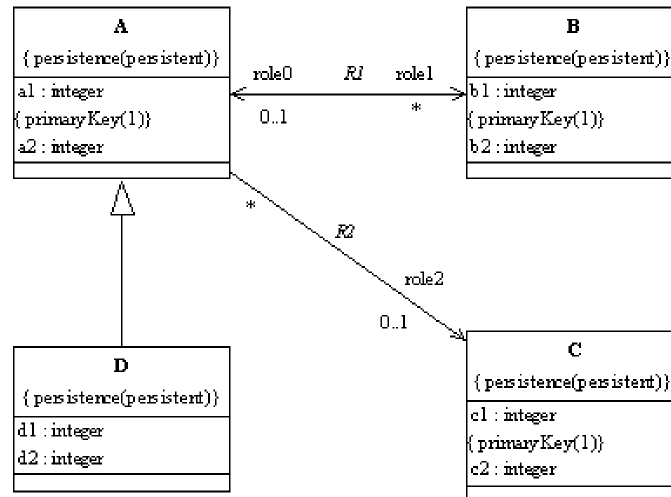


Figure 4-31. Example 3: Generalization and associations

Let us take the example of primary keys, respectively b1 for the B class, c1 for C, a1 for A and D, where the generalization is implemented by one table per concrete class, we will obtain the following associations (Figure 4-32).

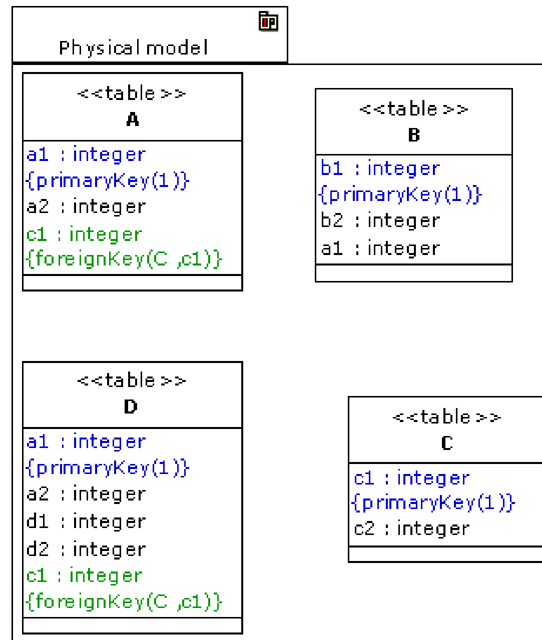


Figure 4-32. Generalization of associations with "*One table per concrete class*" pattern

Textually, this gives the following:

```
A(a1,a2,c1)
B(b1,b2,a1)
C(c1,c2)
D(a1,a2,d1,d2,c1) the external key role2_c1 is
«specialized»!
```

If the generalization had been mapped one table per class, the following would have been obtained (Figure 4-33).

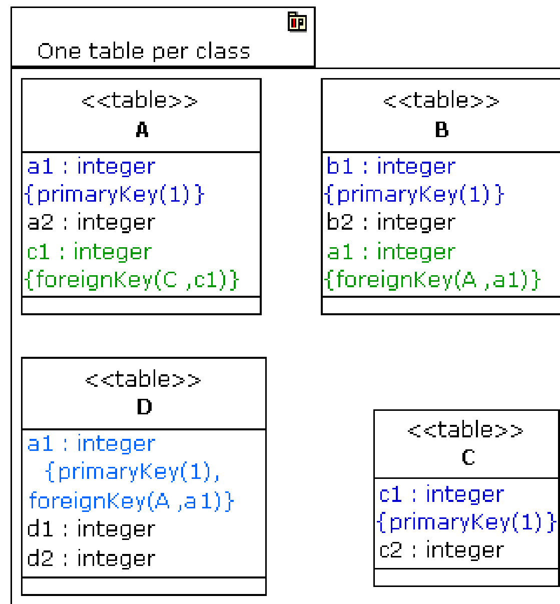


Figure 4-33. Generalization of associations with "*One table per class*" pattern

Textually, this would have given the following:

```
A(a1,a2,c1)
B(b1,b2, a1)
C(c1,c2)
D(a1,d1,d2) only the primary key is common to D and A.
```

If the generalization is mapped with "*One table per generalization tree*", the following would have been obtained (Figure 4-34).

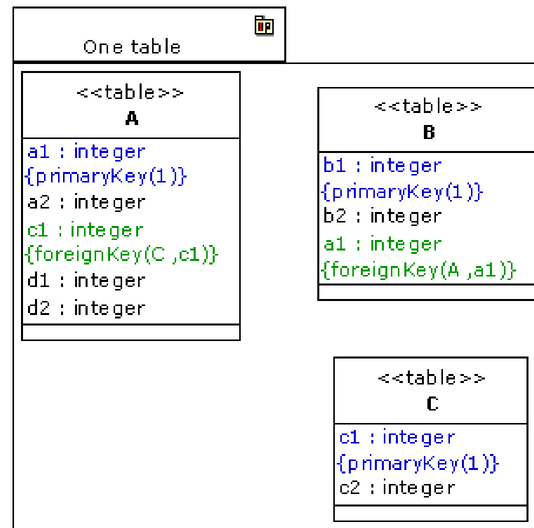


Figure 4-34. Generalization of associations with "*One table per generalization tree*" pattern

Textually, this would have given the following:

```
A(a1,b1,c1,d1,d2)
B(b1,b2,a1)
C(c1,c2)
```

Structure in packages

Overview

Structuring in model packages does not entail any restriction of the database diagram generation. In particular, a database diagram can be modeled using several model packages.

Packages stereotyped <<schema>>

It is possible to group together several relational tables into a relational diagram. This grouping is carried out using the <<*schema*>> (package name) stereotype, which can be attached to a model package or to a class. This stereotype allows the creation of relational tables in a specific relational diagram. Access to these tables is then carried out with the complete name of the table: *diagram name of the table*.

Classes belonging to a package stereotyped <<*schema*>> are generated in a sub-package of the physical model, which is also stereotyped <<*schema*>>.

Components stereotyped <<database>>

In an environment of distributed databases, it is possible to specify the name of the database. This information is available with the <<*database*>> component, which can reference several model packages or classes. The generated SQL linked to the referenced elements allows "*distant*" requests to other databases accessible via a local network.

The tables of the referenced classes are generated in a sub-package stereotyped <<*database*>>.

Integrity constraints

Uniqueness constraint

Definition: Each time an n-tuple is created, the uniqueness constraint makes sure the value of the primary key does not already exist.

The preservation of the uniqueness of a table's primary key is an integrity constraint which has already been dealt with using a "*unique index*".

Referential integrity

Definition: Inter-table constraint, which consists of the imposition that the value of a group of attributes of a first table should appear as key value in a second table.

This integrity constraint, which generally only applies to associations, specifies that an association instance can only exist if the entities which take part in this association already exist. It is set up through the re-definition of *outside keys* that have an impact on the updating operations. When an association element is inserted, it is necessary to check that the referenced elements do exist. Similarly, when a referenced element is deleted, it is necessary to check that the element no longer exists in the association (or, depending on the case, to destroy the corresponding association instance).

Example

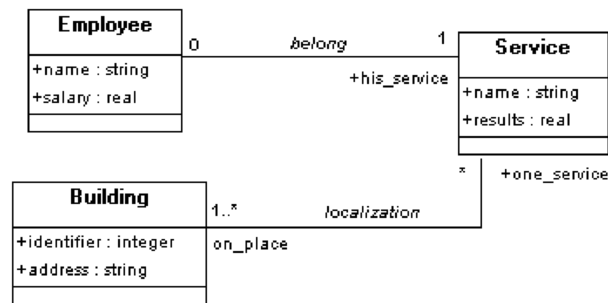


Figure 4-35. Example of associations with integrity control

The mapping of the "*belong*" and "*localization*" associations introduces several reference integrity constraints:

- any value of the *HIS_SERVICE* attribute appearing in the *EMPLOYEE* table should be described in the *SERVICE* table
- any tuple of the *BUILDING_LOCALIZATION_SERVICE* and *SERVICE_LOCALIZATION_BUILDING*, must reference a tuple of the *BUILDING* table and a tuple of the *SERVICE* table

Consequences

Referential integrity constraints are used to check consistency during extraction from and insertion into the table that represents the association.

A ban is imposed during the deletion of an instance referenced by an outside key, or during insertion into a table representing an association of keys which references instances which do not exist. The implementation of these constraints depends on the RDBMS used.

The effect of the integrity constraints can be modified using tagged values specific to the RDBMS, such as {*onDeleteCascade*}.

Generalization and constraints

Between classes which have children, constraint generation on association tables differs according to the chosen generalization mode:

- one table per concrete class
- one table per class

One table per concrete class

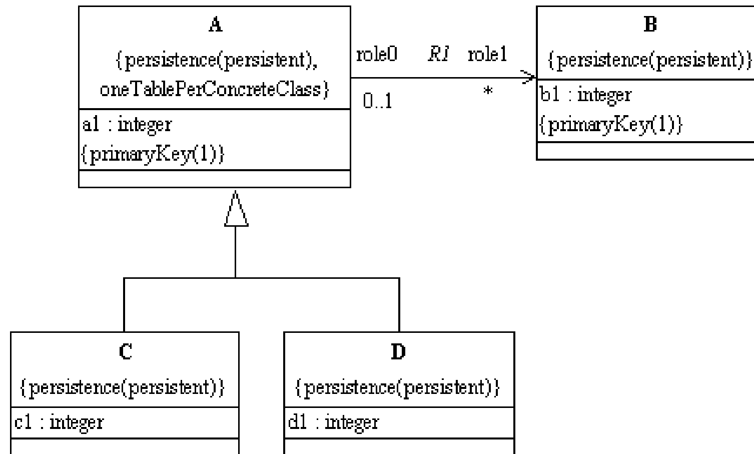


Figure 4-36. One table per concrete class

The SQL tables generated are as follows (Figure 4-37):

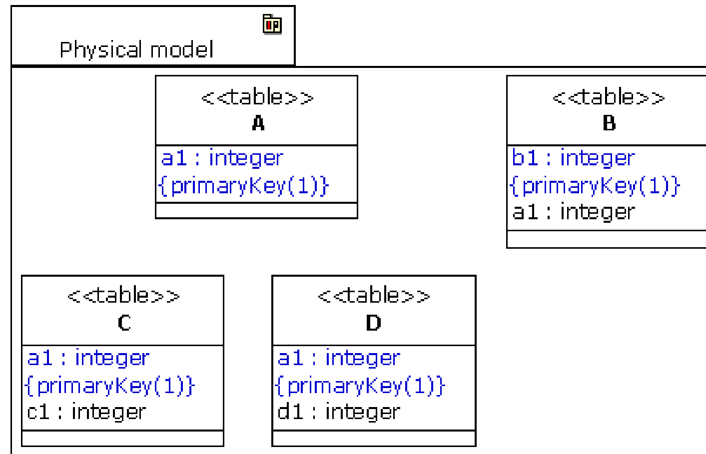


Figure 4-37. The SQL tables generated with one table per concrete class

Textually, this gives the following:

```
A(a1)
B(b1)
C(a1, a2, c1)
D(a1, a2, d1)
role0_of_B(b1, role0_a1)
```

For the "*B*" table, the "*a1*" attribute can correspond to the "*a1*" attribute of the "*A*", "*C*" or "*D*" table.

The module cannot, therefore, generate constraints towards three different tables. The module will add no {*foreignKey*} tagged values to the "*a1*" column, and will generate no constraints for the "*B*" table.

One table per class

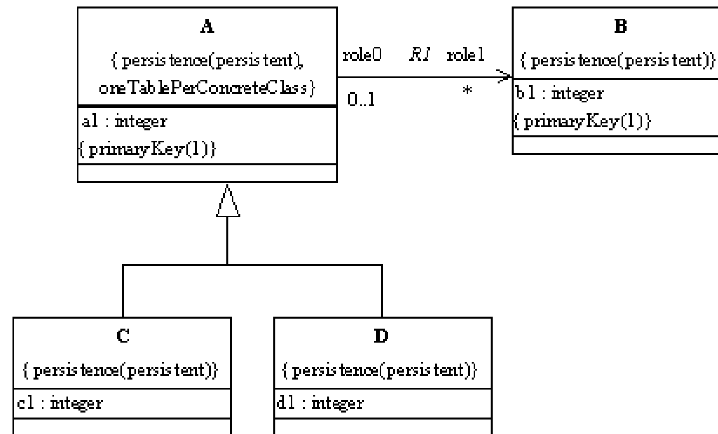


Figure 4-38. One table per class

The SQL tables generated are as follows (Figure 4-39):

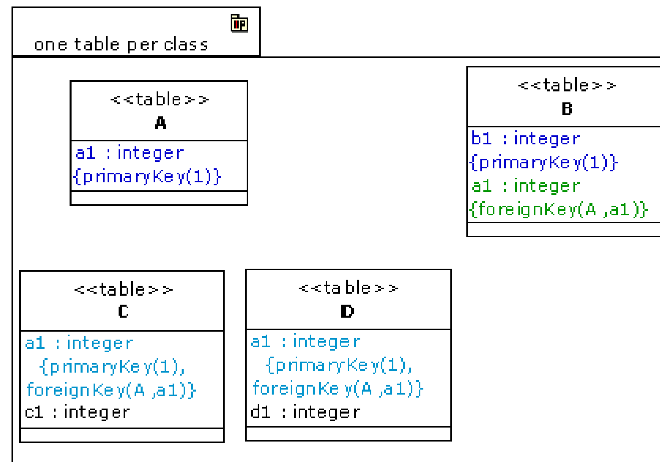


Figure 4-39. The SQL tables generated with one table per class

Textually, this gives the following:

```
A(a1)
B(b1, a1)
C(a1, c1)
D(a1, d1)
```

In one table per class, the module adds a {*foreignKey(A,a1*) tagged value to the a1 attribute of the B table, which generates an integrity constraint. It is sure that for every recording in the C and D tables, a recording must exist in table A with the same primary key.

Constraints generated for "*role0_ofB*" will be as follows:

```
ALTER TABLE B ADD (
   CONSTRAINT role0_of_B_DFK FOREIGN KEY (a1)
      REFERENCES A (a1)
);
```

"Not null" attributes

In addition to the "not null" deduced from the model (constraints on keys, association multiplicity ...), the user may define locally that a particular attribute must always be entered in a database, using the <<notNull>> constraint.

Integrity tagged values

The code generated ensures the database is complete, but if it must be used by other sources, specific reflexes have to be added, such as *on delete cascade* (see the annex specific to the RDBMS used).

Standardized logical schema

Definition

The standardization of a relational database conceptual diagram is generally based on only the first three Normal Forms (NF).

The first three NF have the following definition:

- (1NF) An association is called standardized, or in first normal form, if and only if each of the domains concerned in the association contains only atomic elements.
- (2NF) The association must be in 1 NF and ensure for its primary key total functional dependency of each of its non-key attributes (an fd is called total of A in comparison to B, if no part of B - if B is composed ! - is sufficient to maintain the fd).
- (3NF) The association must be in 2NF and ensure with regard to its primary key, a direct functional dependency (i.e. non transitive) of each of its non key attributes.

Standardization

The *Objecteering/SQL Designer* module checks that the first normal form is respected. Indeed, "*Set of...*" attributes are not accepted.

The user must respect the 2 and 3 normal forms when annotating with the *{primaryKey}* tagged value.

Additional annotations

The {persistence} tagged value

Attributes, associations or generalization links can, where necessary, be annotated locally as non-persistent by the user. The *{persistence}* tagged value is used in this case.

Example

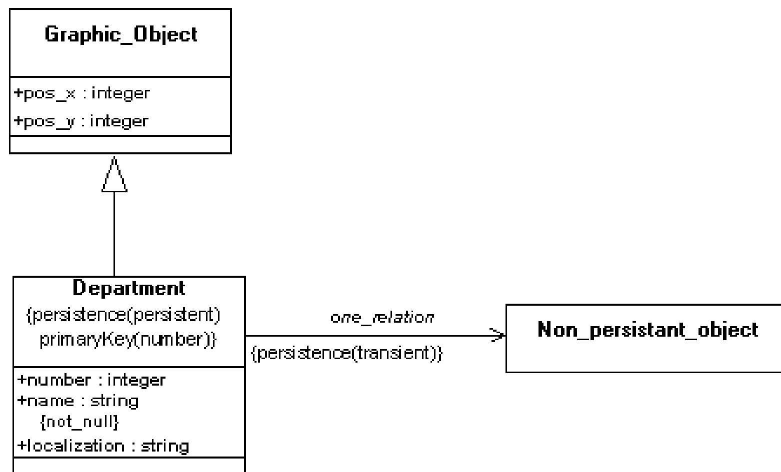


Figure 4-40. Example

in which:

```
CREATE DISTRICT TABLE
      NUMBER...,
      NAME...,
      LOCATION...
);
```

Note: The *{persistence(transient)}* tagged value will be simple and is attached to the non-persistent element for attributes and associations. Classes specialized in a non-persistent way are specified by a parameterized tagged value (with the same name {*persistence(transient)*}), but are located at the level of the child class.

Indexes

Indexes can be defined using the *{indexName(index_name)}* tagged value on a class and the *{indexKey(index_name, rank}* tagged value on each attribute of the index. The advantage of this (in addition to the documentation aspect) is the automatic generation of secondary indexes (indexes which are not unique), used to optimize access to data. An index can either be defined on a class in the analysis model or on a table in the physical model.

Secondary composite keys

Secondary composite keys are used to define unique keys as well as the primary key. For this, the {*keyName(key_name*)} tagged value is used on a class, and the {*compositeKey(key_name, rank)*} tagged value on each of the class' attributes.

Chapter 5: User interface

Interactive interface

Overview

The user interface of generation for relational databases allows you:

- to annotate your model, through the properties editor and the dialog boxes
- to generate a physical model from a logical model
- to update a physical model from a modified logical model
- to generate the SQL which corresponds to the physical model
- to visualize the SQL produced
- to launch the interpretation of the produced SQL

UML modeling project configuration

Before executing any operations concerning the generation of the relational database, you must configure your UML modeling project in order to:

- take into account the UML profile corresponding to the target RDBMS you are using
- choose the default generation characteristics you wish to use

For further information, please refer to the chapter 7, "*Module configuration*", of this user guide.

Outlook

The interface for generating the database diagram provides all or part of the predefined characteristics at UML modeling project level, depending on the case. In this way, default interactive generation corresponds to the planned configuration. However, it is possible to modify these choices dynamically, in order to:

- test different characteristics
- carry out updates on the existing diagram database but without requesting the re-interpretation of the whole of the SQL leading to the final package

Menus

Commands are available by clicking on the right mouse-button.

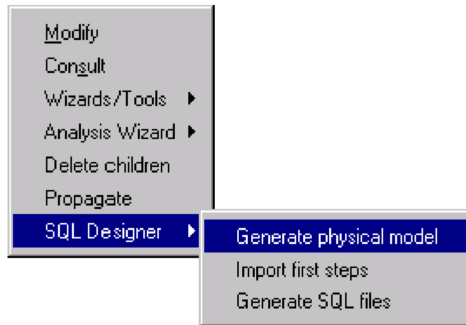


Figure 5-1. Commands available on a package in the Objecteering/UML explorer

The ... command	is used to ...
Generate physical model	launch the generation of the physical model.
Import first steps	import the SQL Designer first steps project.
Generate SQL files	launch the generation of SQL files.

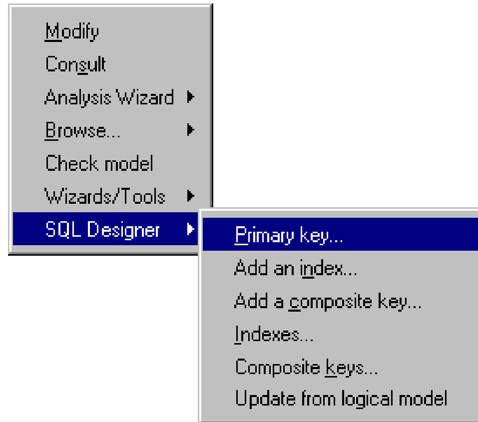


Figure 5-2. Commands available on a class in the Objecteering/UML explorer

The ... command	is used to ...
Primary key	open the primary key window, in which primary keys can be created and managed.
Add an index	open the index creation window, in which indexes can be created.
Add a composite key	open the composite key creation window, in which composite keys can be created.
Indexes	open the index editing window, in which indexes can be managed.
Composite keys	open the composite key editing window, in which composite keys can be managed.
Update from logical model	update a single table of the physical model, without having to update the entire model.

Usage precautions

For each execution of a new SQL, it is recommended that you keep an SQL script after having previously generated it. This will enable you to destroy tables and constraints added to the database. Without this precaution, you may lose table names or constraints always present in the database. Pollution can then occur, or the database can be blocked because of obsolete constraints related to the tables to be destroyed.

For this, the *Objecteering/SQL Designer* module keeps the former SQL delete file, and can launch it automatically before running the creation script. This behavior can be deactivated at module parameter level.

The properties editor and the SQL Designer module

The "SQL" tab of the properties editor on a logical model package

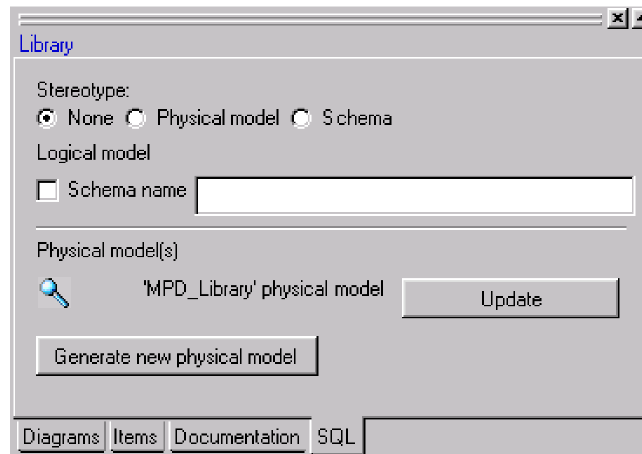


Figure 5-3. The "*SQL*" tab of the properties editor on a logical model package

Key:

- The "*Stereotype*" radio buttons are used to add the <<*schema*>> or <<*sqlPhysicalModel*>> stereotypes to a package.
- The "*Schema name*" field is used to add the {*sqlName*} tagged value to the package. This tagged value defines the name of the schema which corresponds to the physical model, if the package is stereotyped <<schema>>, or the name of the physical model, if a physical model is generated from this package.
- The "*Generate new physical model*" button generates a new physical model from the package.

The "SQL" tab of the properties editor on a physical model package

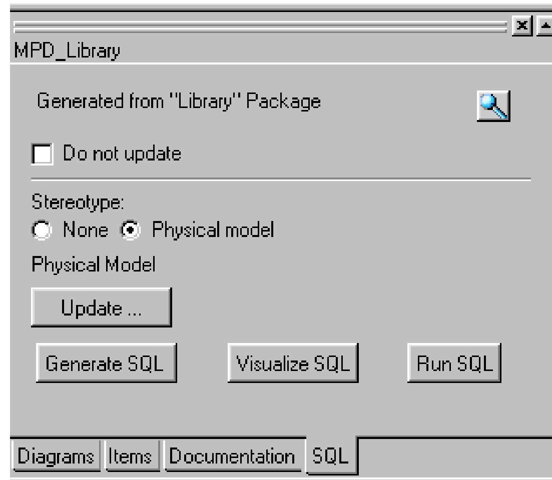


Figure 5-4. The "*SQL*" tab of the properties editor on a physical model package

Key:

- The "*Do not update*" tickbox is used to add the {*sqlDoNotUpdate*} tagged value to the package. This tagged value means that this schema is not updated when the physical model is updated. If this tickbox is checked on a physical model, updating it will do nothing.
- The "*Stereotype*" radio buttons are used to indicate whether or not the model is physical. By default, the "*Physical model*" button is selected.
- The "*Update*" button is used to update the physical model from a package.
- The "*Generate SQL*" button is used to generate the SQL corresponding to the physical model.
- The "*Visualize SQL*" button is used to generate, if necessary, the SQL corresponding to the physical model, and visualize the creation file in a window.
- The "*Run SQL*" button is used to generate, if necessary, the SQL corresponding to the physical model, and run the creation script using the SQL script interpreter of the target RDBMS. If it is not the first generation and if the "*Keep old deletion file*" and "*Execute old deletion file before creation*" parameters have been activated, the old SQL deletion script is executed first.

The "SQL" tab of the properties editor on a package inside a physical model

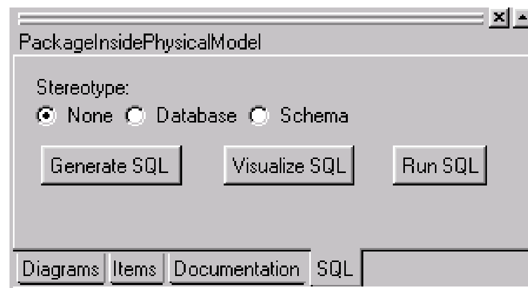


Figure 5-5. The "*SQL*" tab of the properties editor on a package inside a physical model

Key:

- The "*Stereotype*" radio buttons are used to add the <<*schema*>> or <<*database*>> stereotypes to a package inside a physical model. In a physical model, a package can only be stereotyped <<*schema*>>.
- The "*Generate SQL*" button is used to generate the SQL corresponding to the physical model package.
- The "*Visualize SQL*" button is used to generate, if necessary, the SQL corresponding to the physical model package, and visualize the creation file in a window.
- The "*Run SQL*" button is used to generate, if necessary, the SQL corresponding to the physical model package, and run the creation script using the SQL script interpreter of the target RDBMS. If it is not the first generation and if the "*Keep old deletion file*" and "*Execute old deletion file before creation*" parameters have been activated, the old SQL deletion script is executed first.

The "SQL" tab of the properties editor on a class

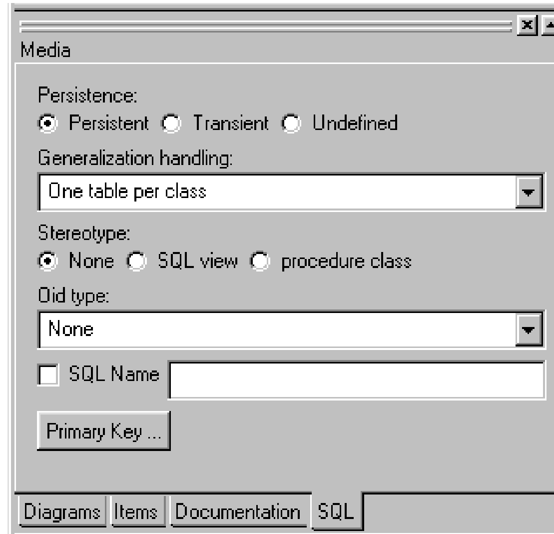


Figure 5-6. The "*SQL*" tab of the properties editor on a class

Key:

- The "*Persistence*" radio buttons are used to add the {*persistence*} tagged value to the class. If the "*Undefined*" radio button is selected, the class will have the same behavior as the parent class.
- The "*Generalization handling*" field is used to select the way in which generalization is managed. "*Undefined (same as parent)*" indicates that the class will handle generalization in the same way as its parent class; "*One table*" adds the {*oneTable*} tagged value to the class; "*One table per class*" adds the {*oneTablePerClass*} tagged value to the class; "*One table per concrete class*" adds the {*oneTablePerConcreteClass*} tagged value to the class.
- The "*Stereotype*" radio buttons are used to stereotype the class <<*sqlView*>> or <<*procedureClass*>>. <<*sqlView*>> means the class is an SQL view, whilst <<*procedureClass*>> indicates that the class is only meant to contain stored procedures.
- The "*oid type*" field is used to select the type of the oid.
- The "*SQL name*" fields are used to add the {*sqlName*} tagged value to the class.
- The "*Primary key...*" button is used to add or remove primary keys to or from the class.

The "SQL" tab of the properties editor on an attribute

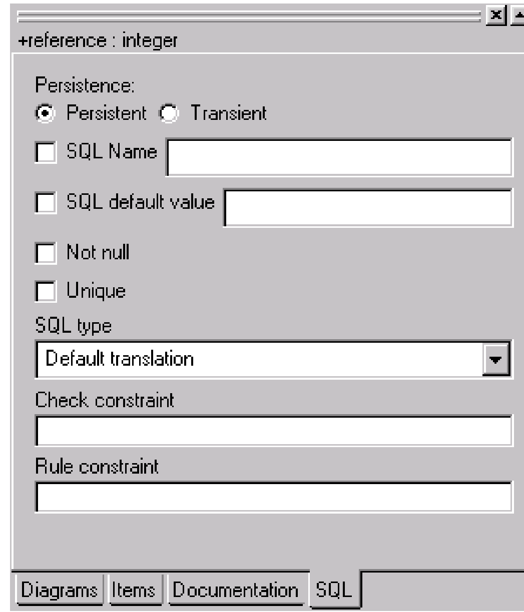


Figure 5-7. The "*SQL*" tab of the properties editor on an attribute

Key:

- The "*Persistence*" radio buttons are used to add the {*persistence*} tagged value to the attribute.
- The "*SQL name*" fields are used to add the {*sqlName*} tagged value to the attribute.
- The "*SQL default value*" fields are used to add the {*sqlDefault*} tagged value to the attribute.
- The "*Not null*" tickbox is used to add the <<*notNull*>> constraint to the attribute.
- The *Unique*" tickbox is used to add the <<*unique*>> constraint to the attribute.
- The "*SQL type*" field adds the {*sqlType(selected element)*} to the attribute, if an element other than "*Default translation*" is selected from the dropdown list. The list of values available depends on the RDBMS selected at module configuration level.
- The "*Check constraint*" field, if defined, adds a <<*check*>> type constraint with the text as its body.
- The "*Rule constraint*" field, if defined, adds a <<*rule*>> constraint, with the entered text as its body.

The "SQL" tab of the properties editor on an association

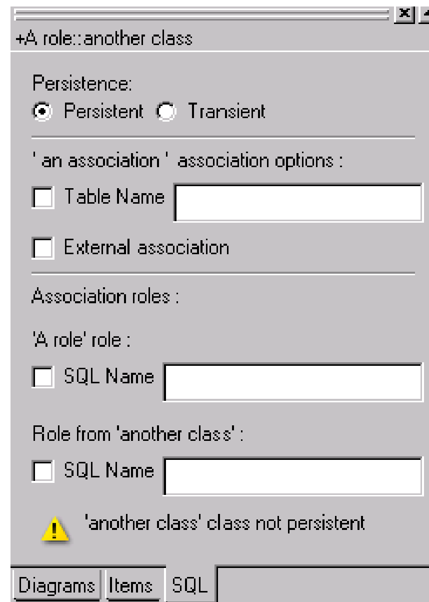


Figure 5-8. The "*SQL*" tab of the properties editor on an association

Key:

- The "*Persistence*" radio buttons are used to add the {*persistence*} tagged value to the attribute.
- The "*X association options*" are the options which apply to the association.
- The "*Table name*" field indicates the name of the table which manages the association, if it is external.
- The "*External association*" tickbox forces the association to be external, by adding the {*external*} tagged value.
- The "*Association roles*" fields are used to add the {*sqlName*} tagged value one or both of the association ends.
- The "*X class not persistent*" warning warns the user that the class connected to the association is not persistent and the model is not consistent (either the association should be transient or the class should be persistent).

For an association which has 0..1 multiplicity, the following option is also available:

- The "*Add "class2" PK to "class1" PK*" tickbox is used to add the {*primaryKey*} tagged value to the "*the_class2*" role.

For an association in which both roles have multiplicity of 1, the following options are also available:

- The "*Merge "class1" into "class2"* " tickbox adds the {*sqlOptimized*} tagged value to the "*the_class1*" role. This signifies that the table which corresponds to the "*class1*" class must be merged in the table which corresponds to the "*class2*" class. The name of the table is then "*class2*".
- Conversely, the "*Merge "class2" into "class1"* " tickbox adds the {*sqlOptimized*} tagged value to the "*the_class2*" role.

The "SQL" tab of the properties editor on a datatype

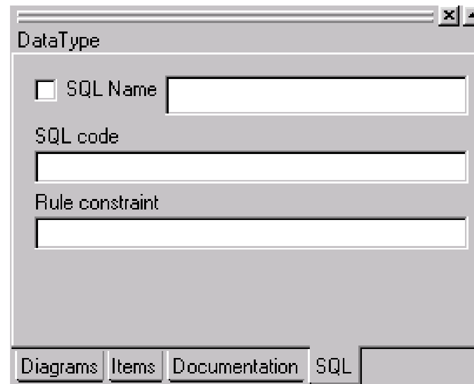


Figure 5-9. The "*SQL*" tab of the properties editor on a datatype

Key:

- The "*SQL name*" tickbox, when checked, adds the {*sqlName*} tagged value to the datatype.
- The "*SQL code*" field is used to add an "*sqlCode*" note to the datatype. This field only appears if the database manager selected at module configuration level is "*Sybase*" or "*SQLServer*".
- The "*Rule constraint*" field, if defined, adds a <<*rule*>> constraint, with the entered text as its body.

The "SQL" tab of the properties editor on a physical model class or a class stereotyped <<table>>

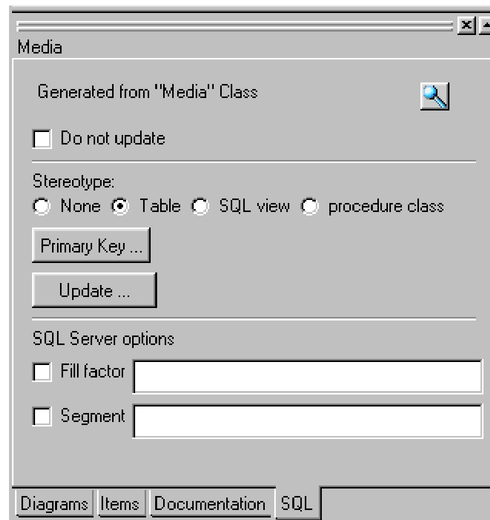


Figure 5-10. The "*SQL*" tab of the properties editor on a physical model class or a class stereotyped <<table>>

Key:

- The "*Stereotype*" radio buttons add, respectively, the <<*table*>> stereotype (the "*SQL table*" button), the <<*sqlView*>> stereotype (the "*SQL view*" button) or the <<*procedureClass*>> stereotype (the "*Procedure class* button).
- The "*Primary key...*" button is used to add or remove primary keys to or from the class.
- The "*Update*" button is used to update a single table of a physical model without updating the whole model.
- The "*SQL Server options*" fields are used to add tagged values specific to the database selected at module configuration level.

The "SQL" tab of the properties editor on a physical model attribute or a table column

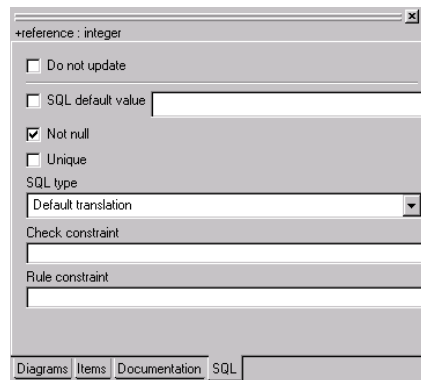


Figure 5-11. The "*SQL*" tab of the properties editor on a physical model attribute or table column

Key:

- The "*Do not update*" tickbox is used to add the {*sqlDoNotUpdate*} tagged value.
- The "*SQL default value*" field is used to add the *{sqlDefaultValue}* tagged value.
- The "*Not null*" tickbox, when checked, adds the <<*notNull*>> constraint to the attribute.
- The "*Unique*" tickbox, when checked, adds the <<*unique*>> constraint to the attribute.
- The "*SQL type*" field is used to add the {*sqlType*} tagged value with the selection as its parameter, as long as the selection is not "*Default translation*".
- The "*Check constraint*" field, if defined, adds a <<*check*>> constraint, with the entered text as its body.
- The "*Rule constraint*" field, if defined, adds a <<*rule*>> constraint, with the entered text as its body.

Generating the physical model

Introduction

There are two ways to generate a physical model:

- by running the "*SQL Designer/Generate physical model*" command available in the context menu which appears when you right-click
- by clicking on the "*Generate new physical model*" button, available in the "*SQL*" tab of the properties editor on a package

Updating a physical model

A physical model can be updated if the logical model to which it is linked has been modified. For this, three solutions are available:

- running the "*SQL Designer/Generate physical model*" command on the physical model, available in the context menu which appears when you right-click. A dialog box informs you that the physical model will be updated. Simply confirm by clicking "*OK*" to start the update.
- clicking on the "*Update*" button in the "*SQL*" tab of the properties editor for a physical model. Simply click on this button to update the physical model.
- clicking on the "*Update*" button in the "*SQL*" tab of the properties editor for a logical model, which is opposite a list of all the physical models generated from this package. Click on this button to start the update of the physical model in question.

Avoiding the update of the physical model

If the user wants a part of the physical model not to be updated, for example, a table or a column added manually, he can annotate this element {*sqlDoNotUpdate*}. The "*Do not update*" tickbox in the "*SQL*" tab of the properties editor can be used to add this tagged value.

Note: If a table is annotated {*sqlDoNotUpdate*}, neither the table nor its contents will be updated.

Generating the SQL

Generated files

On a work product, the "*SQL Designer/Generate SQL files*" command produces a relational diagram. By default, two files are produced. They are located in the directory indicated in the generation work product's dialog box, and are called:

`unit_name_create.sql`

and

`unit_name_delete.sql`

For attributes which have been annotated {*alterTable*}, a third file is generated:

`unit_name_alter.sql`

The file suffix can be parameterized at GUI configuration level.

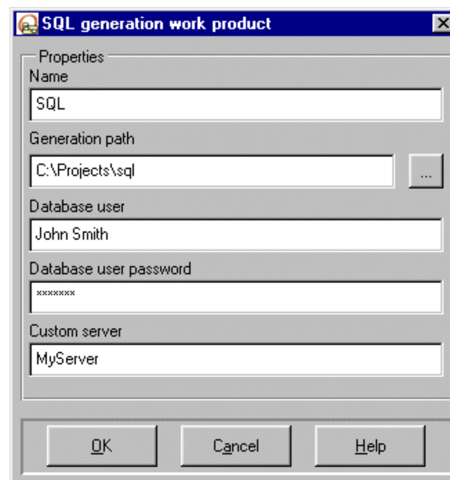


Figure 5-12. The generation work product dialog box

Selection

If generation is activated from a physical model or a package, then it is linked to all the tables belonging to it. Generation cannot be launched from a table.

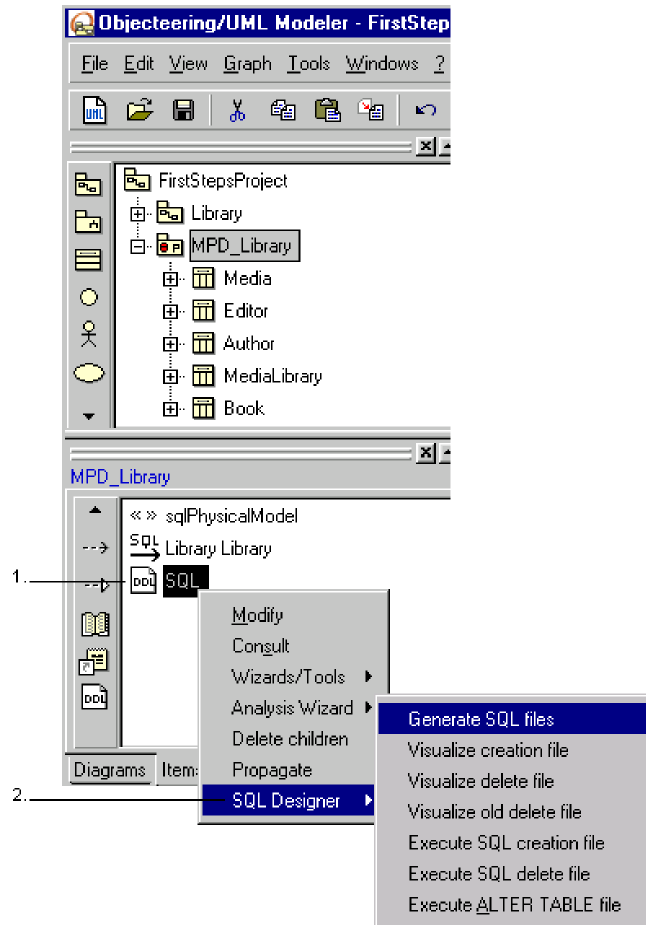


Figure 5-13. Generating the SQL files for the "*MPD_Library*" package

Steps:

1. Click on the SQL generation work product of the "*MPD_Library*" package in the "*Items*" tab of the properties editor using the right mouse-button.
2. Select the "*SQL Designer/Generate SQL files*" commands from the context menu which appears.

Two alternative methods of SQL file generation are also available:

- by clicking on the "*Generate SQL*" button in the properties editor on a package
- by right-clicking on the package and then running the "*SQL Designer/Generate SQL*" commands

Note: Double-clicking on the generation work product runs the "*Propagate*" command for this work product.

Visualizing the SQL

Visualizing the generated SQL

It is possible to visualize the generated SQL file from the properties editor (as shown in Figure 5-14).

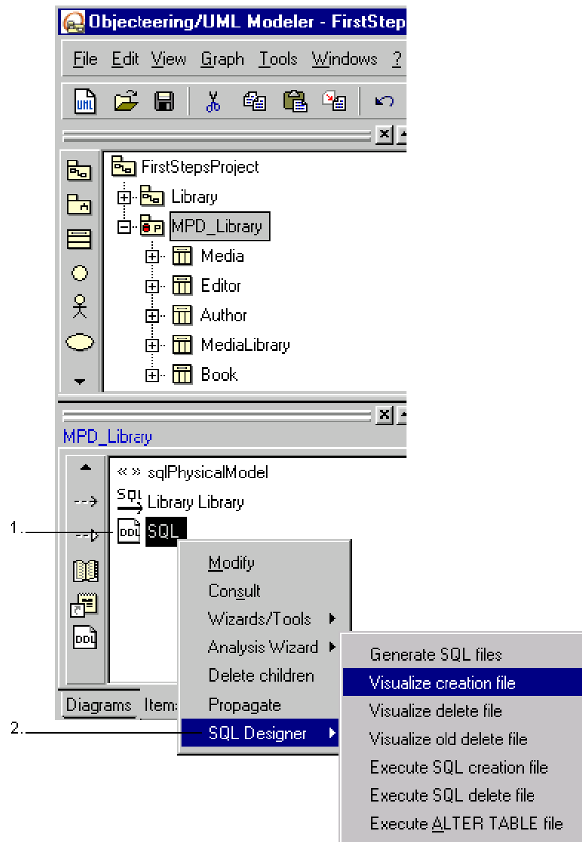


Figure 5-14. Visualizing the SQL file

Steps:

1. Select the "*MPD_Library*" package in the Objecteering/UML explorer.
2. Click on the "*SQL*" generation work product in the "*Items*" tab of the properties editor using the right mouse-button.
3. Select the "*SQL Designer/Visualize creation file*" commands from the context menu which appears.

The "*alter*" file can also be visualized using the "*Visualize creation file*" command.

The "*Visualize delete file*" and "*Visualize old delete file*" commands are used to visualize the delete file and the delete file from a previous generation. This file only exists if the "*Keep old delete file*" module parameter is active.

Generated SQL

The "*Editing the generated file*" window (as shown in Figure 5-15) is used to visualize the element's generated SQL file.

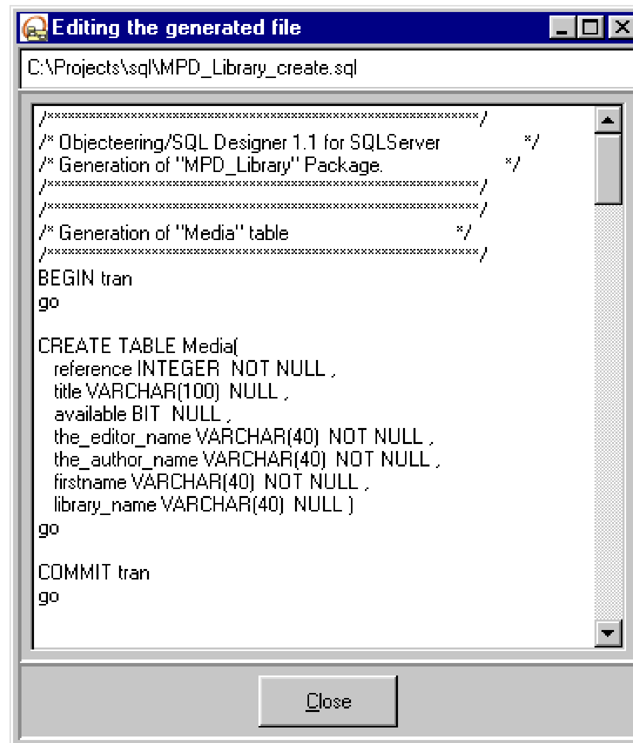


Figure 5-15. SQL generated for the "*MPD_Library*" package

Executing the SQL

Procedure

We are now going to execute the SQL, by following the procedure represented below in Figure 5-16.

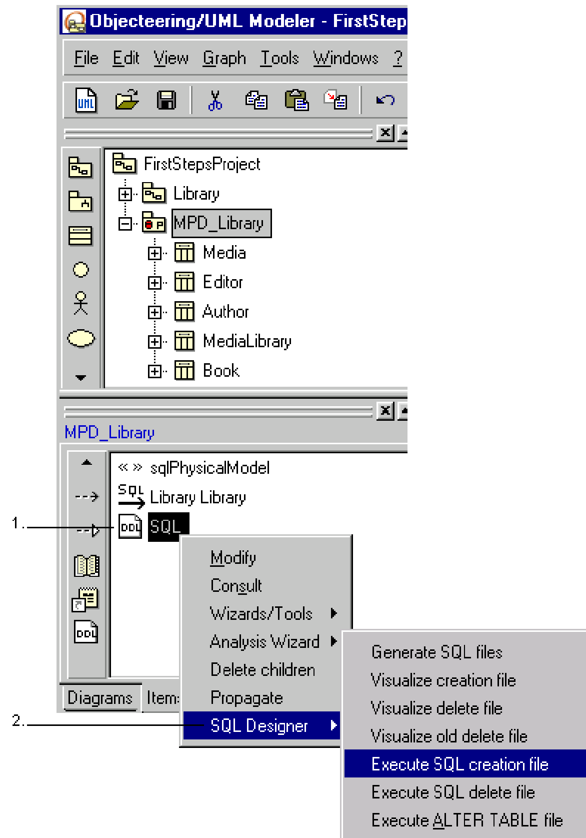


Figure 5-16. Executing the SQL

Steps:

1. Select the generation work product of the "*MPD_Library*" package in the "*Items*" tab of the properties editor using the right mouse-button.
2. Select the "*SQL Designer/Execute SQL creation file*" commands from the context menu which appears.

Example of tables created in a database

The descriptions of the tables created below correspond to the "*MPD_Library*" package.

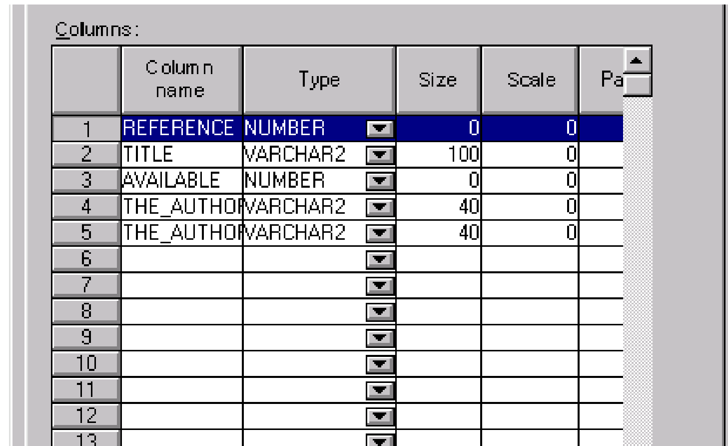


Figure 5-17. Tables created for the "*Media*" class

Executing the alter table file

Where attributes have been annotated using the {*alterTable*} tagged value, *Objecteering/SQL Designer* generates, in addition to the normal SQL files, a file named:

```
unit_name_alter.sql
```

To execute this file, the "*Execute alter table file*" command is used (as shown in Figure 5-18).

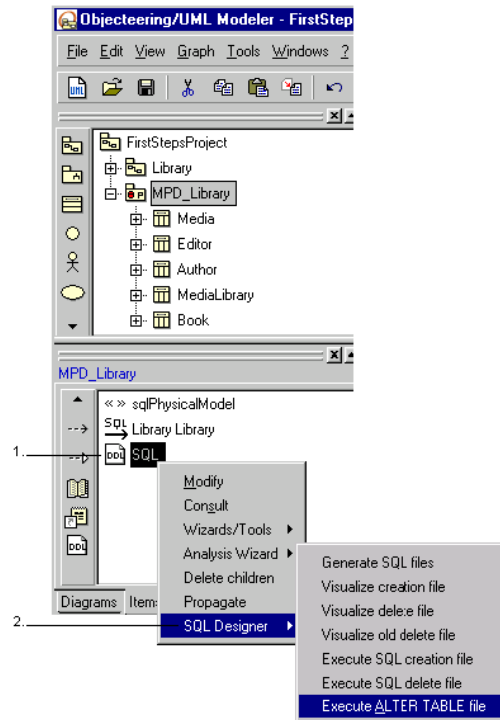


Figure 5-18. Running the "*Execute alter table file*" command

Steps:

1 - Select the generation work product of the "*MPD_Library*" package in the "*Items*" tab of the properties editor using the right mouse-button.

2 - Select the "*SQL Designer/Execute alter table file*" commands from the context menu which appears.

Chapter 6: Annotating the model

Tagged value types

Introduction

The tagged values provided by Objecteering/UML are used to adapt *UML Profile for SQL* semantics to a specific UML model, in order to generate *UML Profile for SQL* notions accurately.

Tagged values on an association

Tagged value	SQL	RDBMS
{external}		
{cluster(cluster_name)}	CLUSTER	Oracle
{tablespace(tablespace_name)}	TABLESPACE (Oracle) SEGMENT (Sybase)	
{partition(partition_name)}		Sybase only
{pctfree(value)}	PCTFREE	Oracle
{pctused(value)}	PCTUSED	Oracle
{initrans(value)}	INITRANS	Oracle
{maxtrans(value)}	MAXTRANS	Oracle
{storage(value)}	STORAGE	Oracle
{maxRowsPerPage(value)}		Sybase
{fillfactor(value)}		

Tagged values on an association end

Tagged value	SQL
{sqlOptimized}	
{onDeleteCascade}	
{primaryKey}	

Tagged values on an attribute

Tagged value	SQL
{sqlDefault(default_value}	Default
{globalTable(table_name)}	
{indexKey(index_name,rank)}	
{primaryKey(rank)}	
{foreignKey(table_name,column_name [constraint_name])}	
{compositeKey(key_name,rank)}	
{persistence(type_persistence)}	
{sqlType(specific_type)}	
{alterTable}	

Tagged values on a class

Tagged value	SQL
{fillfactor(value)}	WITH FILLFACTOR (Sybase)
{maxRowsPerPage(value)}	WITH MAX_ROWS_PER_PAGE (Sybase)
cluster_name : {cluster(cluster_columns,...)}	CLUSTER (Oracle)
{initrans(value)}	INITRANS (Oracle)
{maxtrans(value)}	MAXTRANS (Oracle)
{pctfree(value)}	PCTFREE (Oracle)
{pctused(value)}	PCTUSED (Oracle)
{storage(clause)}	STORAGE (Oracle)
{indexSpace(tablespace_name)}	TABLESPACE (Oracle)
{indexStorage(clause)}	STORAGE (Oracle)
{classAttributesTableName(table_name)}	
{oneTablePerConcreteClass}	
{oneTablePerClass}	
{oneTable}	
{noDDL}	
{oid}	
{tablespace(tablespace_name)}	TABLESPACE (Oracle), ON (Sybase)
{partition(number)}	PARTITION (Sybase)
{keyName(key_name)}	
{indexName(index_name)}	
{persistence(type_persistence)}	

Tagged values on a component

Tagged value	SQL
{dbName(database_path)}	

Tagged values on a generalization

Tagged value	SQL
{persistence(the_persistence)}	

Tagged values on a model element

Tagged value	SQL
{sqlName(SQL_Name)}	

Tagged values on an operation

Tagged value	SQL
{tgInsert}	
{tgUpdate}	
{tgDelete}	
{tgBefore}	
{tgAfter}	

Note: The tagged values detailed in the table above must only be used on operations which are stereotyped <<*trigger*>>.

Note types

Introduction

Objecteering/UML note types are used to complete the UML model with texts in *UML Profile for SQL* syntax.

Note types on a class

The ... note type	is used to...
sqlBefore	indicate the SQL code to be added to the .SQL file before the SQL code corresponding to the modeled class.
sqlAfter	indicate the SQL code to be added to the .SQL file after the SQL code corresponding to the modeled class.

Note types on a datatype

The ... note type	is used to...
sqlCode	provide the SQL definition of the datatype (Sybase).

Note types on an operation

The ... note type	is used to...
sqlCode	indicate the SQL code of an operation, so as to allow the entry of the code for a stored procedure or a trigger.

Note types on a package

The ... note type	is used to...
sqlBefore	define the SQL code which will be added at the beginning of the .SQL file associated with the modeled package.
sqlAfter	indicate the SQL code which will be added to the end of the .SQL file associated with the modeled package.

Stereotypes

Introduction

Objecteering/UML defines stereotypes, used to designate certain objects as being concerned by the generation of *UML Profile for SQL* code. In this way, a class concerning an error which has occurred will be treated as an exception, if it bears the <<*XXX:exception*>> stereotype.

Stereotypes on a constraint

The ... stereotype	is used to...	SQL
<<notNull>>	indicate that the column must have a value.	NOT NULL
<<null>>	indicate the column may not have a value.	NULL
<<unique>>	Indicate that the values taken by the attribute must be unique for the entire table.	UNIQUE
<<check>>	associate a "*check*" type constraint to an attribute.	CHECK
<<rule>>	associate a "*rule*" type constraint to an attribute (Sybase).	CREATE RULE

Stereotypes on a class

The ... stereotype	is used to...	SQL
<<procedureClass>>	specify that the class only contains stored procedures.	
<<table>>	indicate that the class is an SQL table.	
<<sqlView>>	describe an SQL view.	

Stereotypes on a component

The ... stereotype	is used to...	SQL
<<database>>	indicate the creation of a database. The implementation of components is used to model dependencies between databases, and their deployment.	

Stereotypes on an operation

The ... stereotype	is used to...	SQL
<<createView>>	indicate that the operation contains SQL code, placed on an "*sqlcode*" note, allowing a view to be created.	
<<trigger>>	indicate that the operation is a trigger.	CREATE TRIGGER
<<storedProcedure>>	indicate that the operation is a stored procedure.	CREATE PROCEDURE

Stereotypes on a package

The ... stereotype	is used to...	SQL
<<schema>>	indicate that the package is a schema.	
<<database>>	indicate that the package is a database. This stereotype is only used in the physical model.	
<<sqlPhysicalModel>>	indicate that the package is a physical model.	

Annotating an association

Tagged values on an association

The tagged values used to annotate associations are presented in the following table:

The ... tagged value	Role	SQL
{external}	Indicates that the association should be managed by a specific table, whatever its multiplicity.	
{cluster(cluster_name)}	Links the table to a predefined cluster.	CLUSTER
{tablespace(tablespace_name)}	Indicates the name of the "*tablespace*" for Oracle or the name of the "*segment*" for Sybase, used during table creation.	TABLESPACE (Oracle) SEGMENT (Sybase)
{partition(partition_name)}	Used to parameterize the physical model table, which manages this external association.	
{pctfree(value)}	Used to parameterize the physical model table, which manages this external association.	PCTFREE
{pctused(value)}	Used to parameterize the physical model table, which manages this external association.	PCTUSED
{initrans(value)}	Used to parameterize the physical model table, which manages this external association.	INITRANS
{maxtrans(value)}	Used to parameterize the physical model table, which manages this external association.	MAXTRANS

The ... tagged value	Role	SQL
{storage(value)}	Used to parameterize the physical model table, which manages this external association.	STORAGE
{maxRowsPerPage(value)}	Used to parameterize the physical model table, which manages this external association.	
{fillfactor(value)}	Used to parameterize the physical model table, which manages this external association.	

The {external} tagged value

The {external} tagged value is used to force the creation of an external table, and applies to those associations normally included (in other words, whose maximum multiplicity is 1).

The {cluster} tagged value

The {cluster} tagged value is used to link the table to a predefined cluster.

The {tablespace} tagged value

The {tablespace} tagged value is used to specify the logical storage unit of the data.

The {partition} tagged value

The {partition} tagged value is used to specify the number of pages linked together in the table.

The {pctfree} tagged value

The {pctfree} tagged value is used to change the table's characteristics.

The {pctused} tagged value

The {pctused} tagged value is used to change the table's characteristics.

The {initrans} tagged value

The {initrans} tagged value is used to change the table's characteristics.

The {maxtrans} tagged value

The {maxtrans} tagged value is used to change the table's characteristics.

The {storage} tagged value

The {storage} tagged value is used to change the table's characteristics.

The {maxRowsPerPage} tagged value

The {maxRowsPerPage} tagged value is used to specify the maximum number of tuples per page.

The {fillfactor} tagged value

The {fillfactor} tagged value is used to specify the percentage of pages taken up by the index.

Annotating an association end

Tagged values on an association end

The tagged values used to annotate association ends are presented in the following table:

The ... tagged value	Role	SQL
{sqlOptimized}	Indicates that the source table should be merged with the destination table (1-1 association). For class associations or 0..1 or 1..1 associations, it indicates that the class association's attributes are placed in the opposite class, without generating intermediate tables.	
{onDeleteCascade}	Indicates that foreign keys must be destroyed if the primary key is deleted, and provides the ON DELETE CASCADE option for Oracle and an ON DELETE trigger for Sybase.	
{primaryKey}	Adds the primary key of the destination class to the primary key of the source class (as for a composition association).	

The {sqlOptimized} tagged value

For an association with 1-1 multiplicity, this tagged value indicates that the source table must be merged with the destination table.

For a class association with 1-1 or 1-* multiplicity, the attributes of the class association will be placed in the destination class of the association end annotated {sqlOptimized}, without generating an intermediary table.

The {onDeleteCascade} tagged value

The {onDeleteCascade} tagged value is used to maintain repository integrity constraints on associations. For further information, please refer to the "*Association annotations*" section in chapter 9 of this user guide.

The {primaryKey} tagged value

The {primaryKey} tagged value adds the primary key of the destination class to the primary key of the source class (as for a composition association).

Annotating an attribute

Tagged values on an attribute

The tagged values used to annotate attributes are presented in the following table:

The ... tagged value	Role	SQL
{sqlDefault(default_value}	Used to specify a default value for the attribute.	
{globalTable(table_name)}	Indicates that the attribute will be generated in a global table (this applies only to class attributes).	
{indexKey(index_name,rank)}	Indicates that the attribute participates in the definition of an index.	
{primaryKey(rank)}	Indicates that the attribute is the primary key of the table in which the attribute is defined. This tagged value contains one parameter, used to specify the order of the field, if the primary key is made up of several attributes.	
{foreignKey(table_name,column_name [constraint_name])}	This indicates that the attribute is a foreign key defined on a table.	
{compositeKey(key_name,rank)}	Indicates that the attribute participates in the definition of an index. This tagged value has two parameters, used to specify the name of the index, and the order in which the attribute will be defined in this index. This type of index must be defined for a table.	
{persistence(type_persistence)}	Indicates in the analysis model whether or not the attribute is persistent.	

The ... tagged value	Role	SQL
{sqlType(specific_type)}	Allows the used of specific types, such as LONG, RAW, LONG RAW, LONG VARCHAR, and so on.	
{alterTable}	Indicates to the module that it should generate a "u*nit_name_alter.sql*" file for each table containing attributes annotated {alterTable}	

The {sqlDefault} tagged value

The {sqlDefault} tagged value indicates the default value in the SQL table. This tagged value is used to express a default value which is not equal to that of the standard "*Expression of value*" field.

The {globalTable} tagged value

The {globalTable} tagged value is only used on class attributes. It indicates whether the class attribute should be generated in a global table, which groups together all class attributes. The structure of this table is explained in "*Class attributes*" paragraph of the "*Attributes*" section in chapter 4 of this user guide.

This tagged value's parameter specifies the name of the global table. If this tagged value is not defined, the attribute will be generated in the table specific to the class attributes of the class.

The {indexKey} tagged value

The {indexKey} tagged value indicates that the attribute participates in the definition of an index. This tagged value contains two parameters - the name of the index, and the order in which the attribute will be defined in this index. The class or the table must be annotated {indexName(index_name)}.

The {primaryKey} tagged value

The {primaryKey} tagged value indicates that the attribute is the primary key of the table in which the attribute is defined. This tagged value contains one parameter, used to specify the order of the field in the primary key.

The {foreignKey} tagged value

The {foreignKey} tagged value indicates that the attribute is a foreign key defined on a table. Two parameters are necessary with this tagged value - the name of the class containing the referenced attribute, and the name of the attribute itself. The third parameter is only necessary if more than one foreign key constraint exists between the same two tables.

The {compositeKey} tagged value

The {compositeKey} tagged value indicates that the attribute participates in the definition of a composite key. This tagged value has two parameters - the name of the key, and the order in which the attribute will be defined within this key. The class or the table must be annotated {keyName(key_name)}.

The {persistence} tagged value

The {persistence} tagged value is used to specify whether or not the attribute is persistent. If its parameter is (persistent), then the attribute is persistent and has an equivalent in the physical model (a column). If the parameter is (transient), then the attribute is not persistent.

The {sqlType} tagged value

The {sqlType} tagged value indicates that the module should use a specific type given as a parameter. Available types differ according to the RDBMS selected at module parameter level. It is possible to customize generation, by modifying existing types or creating new types in the types package.

The {alterTable} tagged value

The {*alterTable*} tagged value is used to indicate that the module should generate a "*unit_name_alter.sql*" file for every existing table which contains attributes annotated in this way.

Annotating a class

Tagged values on a class

The tagged values used to annotate classes are presented in the following table:

The ... tagged value	Role	SQL
{fillfactor(value)}	Specifies the percentage of pages filled by the index.	WITH FILLFACTOR (Sybase)
{maxRowsPerPage(value)}	Specifies the maximum number of tuples per page.	WITH MAX_ROWS_PER_PAGE (Sybase)
cluster_name : {cluster(cluster_columns,...)}	Links the table to a predefined cluster.	CLUSTER (Oracle)
{initrans(value)}	Changes the characteristics of the table.	INITRANS (Oracle)
{maxtrans(value)}	Changes the characteristics of the table.	MAXTRANS (Oracle)
{pctfree(value)}	Changes the characteristics of the table.	PCTFREE (Oracle)
{pctused(value)}	Changes the characteristics of the table.	PCTUSED (Oracle)
{storage(clause)}	Changes the characteristics of the table.	STORAGE (Oracle)
{indexSpace(tablespace_name)}	Specifies the tablespace for the index.	TABLESPACE (Oracle)
{indexStorage(clause)}	Specifies the STORAGE class for creating the index.	STORAGE (Oracle)
{classAttributesTableName(table_name)}	Defines the name of the table which will contain all the class attributes, except those annotated {globalTable}.	
{oneTablePerConcreteClass}	Indicates table mapping.	

The ... tagged value	Role	SQL
{oneTablePerClass}	Indicates table mapping.	
{oneTable}	Indicates table mapping.	
{oid}	Indicates that an attribute of "oid" type should be generated in the table corresponding to this class.	
{tablespace(tablespace_name)}	Specifies the logical storage unit of the data.	TABLESPACE (Oracle), ON (Sybase)
{partition(number)}	Specifies the number of pages linked together in the table.	PARTITION (Sybase)
{keyName(key_name)}	Defines "unique" constraints.	
{indexName(index_name)}	Declares an index associated to a table.	
{persistence(type_persistence)}	Indicates in the analysis model whether or not the class is persistent.	

The {fillfactor} tagged value

The {fillfactor} tagged value specifies the fill factor of each index page. This must be between 1 and 100. If the fill factor is not defined, its default value is 0. A 0 fill factor creates indexes clustered with full pages and indexes which are not clustered by full page sheets.

Note: For further information, please refer to volume 1 of Sybase SQL Server Reference Manual, page 3-79.

The {maxRowsPerPage} tagged value

The {maxRowsPerPage} tagged value is used to specify the maximum number of tuples per page in the table. This must be between 0 and 256. If it is not defined, its default value is 0. A 0 value creates indexes that are clustered with full pages and indexes which are not clustered by full page sheets.

Note: For further information, please refer to volume 1 of Sybase SQL Server Reference Manual, page 3-80.

The {cluster} tagged value

The {cluster} tagged value is used to link the table to a pre-existing cluster (for example, one created within an "*sqlbefore*" item).

The descriptor of this tagged value provides the cluster's name, whilst the parameters correspond to the list of columns associated with the cluster.

The {initrans} tagged value

The {initrans} tagged value takes as its unique parameter the value of the Oracle parameter, with the same name as is generated in this case. It must not be used with a cluster (for further information, please refer to the Oracle SQL documentation, page 4-210).

The {maxtrans} tagged value

The {maxtrans} tagged value takes as its unique parameter the value of the Oracle parameter, with the same name as is generated in this case. It must not be used with a cluster (for further information, please refer to the Oracle SQL documentation, page 4-210).

The {pctfree} tagged value

The {pctfree} tagged value takes as its unique parameter the value of the Oracle parameter, with the same name as is generated in this case. It must not be used with a cluster (for further information, please refer to the Oracle SQL documentation, page 4-210).

The {pctused} tagged value

The {pctused} tagged value takes as its unique parameter the value of the Oracle parameter, with the same name as is generated in this case. It must not be used with a cluster (for further information, please refer to the Oracle SQL documentation, page 4-210).

The {storage} tagged value

The {storage} tagged value takes as its unique parameter the value of the Oracle parameter, with the same name as is generated in this case. It must not be used with a cluster (for further information, please refer to the Oracle SQL documentation, page 4-210).

The {indexSpace} tagged value

The {indexspace} tagged value is used to specify the "*tablespace*" where the class index will be created.

The {indexStorage} tagged value

The {indexStorage} tagged value is used to specify the "*STORAGE*" Oracle clause for indexes. The parameter of this tagged value must contain, between inverted commas, the different Oracle parameters associated with a parameter's values. For example:

```
"INITIAL 1024 NEXT 2048"
```

The {classAttributesTableName} tagged value

The {classAttributesTableName} tagged value is used to indicate the name of the table which will contain all those class attributes of this class not annotated {globalTable}.

The {oneTablePerConcreteClass} tagged value

The {oneTablePerConcreteClass} tagged value is the default generalization table mapping mode. It associates a table in the physical model with each concrete class.

The {oneTablePerClass} tagged value

The {oneTablePerClass} tagged value is used to specify the generalization table mapping mode. This mode generates, for each modeled class, a table in the physical model. The rules for the previous transformation then apply to all classes, without exception.

The {oneTable} tagged value

The {oneTable} tagged value is used to specify the generalization table mapping mode. This mode generates a single table containing the set of attributes for all the tables which make up a generalization tree.

The {oid} tagged value

The {oid} tagged value is used to indicate that a primary key "oid" attribute must be generated.

If this tagged value has no parameters, the type of this attribute depends on the RDBMS selected in the "*Database*" module parameter (this is ROWID type for Oracle, and INTEGER with the IDENTITY option for Sybase). The user can add an "*integer*" parameter, indicating that the attribute will be an integer, or the "*string*" parameter, indicating that the attribute will be a string.

This tagged value must only be used on logical model classes.

The {tablespace} tagged value

For Oracle, the {tablespace} tagged value is used to specify the logical storage unit of the data. Classes not annotated {tablespace} are stored in the SYSTEM tablespace.

For Sybase, the {tablespace} tagged value is used to specify a specific segment of allocation of the table's pages. The segment will have to have been created first by the adminstrator, using the following procedure:

```
sp_addsegment
```

The {partition} tagged value

The {partition} tagged value specifies the number of chain pages in the table. It allows the SQL to execute insertions at the same time on the last page of each chain. The partition must be greater than or equal to 2. By default, there is only one chain of pages per table.

Note: For further information, please refer to volume 1 of Sybase SQL Server Reference Manual, page 3-15.

The {keyName} tagged value

The {keyName} tagged value allows you to declare a composite key on the table associated with the class. The parameter must contain the composite key name. The composite key content is defined with {keyName} tagged values on the attributes belonging to it. Several composite keys can be declared for the same class.

The {indexName} tagged value

The {indexName} tagged value allows you to declare an index on the table associated with the class. The parameter must contain the index name. The index content is defined with {indexKey} tagged values on the attributes belonging to it. Several indexes can be declared for the same class.

The {persistence} tagged value

The {persistence} tagged value's parameter is used to indicate whether or not the class is persistent. If the parameter is (persistent), the class is persistent and has an equivalent in the physical model (a table). If the parameter is (transient), the class is not persistent.

Notes on a class

The notes used to annotate classes are presented in the following table:

The ... note type	is used to...
sqlBefore	indicate the SQL code to be added to the .SQL file before the SQL code corresponding to the modeled class.
sqlAfter	indicate the SQL code to be added to the .SQL file after the SQL code corresponding to the modeled class.

Stereotypes on a class

The stereotypes used to annotate classes are presented in the following table:

The ... stereotype	is used to...
<<procedureClass>>	specify that the class only contains stored procedures.
<<table>>	indicate that the class is an SQL table.
<<sqlView>>	describe an SQL view.

Annotating a component

Tagged values on a component

The tagged values used to annotate components are presented in the following table:

The ... tagged value	Role	SQL
{dbName(database_path)}	Indicates the physical path of the database (Oracle).	

The {dbName} tagged value

The {dbName} tagged value is used to reference tables defined in another database, and to specify the path in the parameter (ORACLE only).

Stereotypes on a component

The ... stereotype	is used to...
<<database>>	indicate the creation of a database. The implementation of components is used in the logical model to model dependencies between databases, and their deployment. In the physical model, databases are modeled by <<database>> packages.

Annotating a constraint

Stereotypes on a constraint

The stereotypes used to annotate constraints are presented in the following table:

The ... stereotype	is used to...	SQL
<<notNull>>	indicate that the column must have a value.	NOT NULL
<<null>>	indicate the column may not have a value.	NULL
<<unique>>	Indicate that the values taken by the attribute must be unique for the entire table.	UNIQUE
<<check>>	associate a "*check*" type constraint to an attribute.	CHECK
<<rule>>	associate a "*rule*" type constraint to an attribute (Sybase).	CREATE RULE

The <<notNull>> stereotype

The <<notNull>> stereotype is used for constraints on attributes. It indicates that the attribute must be valued.

The <<null>> stereotype

The <<null>> stereotype is used for constraints on attributes. It indicates that the column cannot be valued. This stereotype imposes the generation of a "NULL" constraint in the column corresponding to the attribute. It allows the use of an "undetermined" when inserting a tuple for which data has not been entered in the column.

The <<unique>> stereotype

The <<unique>> stereotype is used for constraints on attributes. It indicates that the values taken by the attribute must be unique for the entire table.

The <<check>> stereotype

The <<check>> stereotype is used for constraints on attributes, and is used to associate a <<check>> type constraint to an attribute.

The <<check>> constraint allows the specification of a Boolean expression that will have to be checked, in order for a tuple to be inserted into the table.

This constraint has as its body the logical condition, used as is, for building the clause's SQL. We recommend that you place this body between inverted commas for the syntactical analyzer. For example:

```
{check} ("(att1 + att2) between 1 and 12")
```

Note: Remember in this case that if you wish to incorporate the " character into the text, you must enter the ~ character before it.

The <<rule>> stereotype

The <<rule>> stereotype is used to associate a "*rule*" type constraint to an attribute (Sybase only).

Annotating a datatype

Notes on a datatype

The notes used to annotate datatypes are presented in the following table:

The ... note type	is used to...
sqlCode	provide the SQL definition of the datatype (Sybase).

Annotating a generalization

Tagged values on a generalization

The tagged values used to annotate generalizations are presented in the following table:

The ... tagged value	Role	SQL
{persistence(the_persistence)}	Indicates whether or not the generalization is persistent.	

The {persistence} tagged value

The {persistence} tagged value is used to indicate whether or not the generalization is persistent. If the parameter is (persistent), then the generalization is persistent, whilst the (transient) parameter indicates that the generalization is not persistent.

Annotating a model element

Tagged values on a model element

The tagged values used to annotate model elements are presented in the following table:

The ... tagged value	Role	SQL
{sqlName(SQL_Name)}	Indicates the model element's SQL name.	

The {sqlName} tagged value

The {sqlName} tagged value is used as follows:

- for a class, it indicates the name of the table, the view
- for an attribute, it indicates the name of the table column
- for an association end, it is the prefix for double foreign key attributes and for foreign key constraint names
- for an association, it indicates the name of the external table
- for an operation, it indicates the SQL name of the trigger or of the stored procedure
- for a datatype, it indicates the SQL name of the type or the definition of the type if no "*sqlCode*" note is present on the datatype
- for a package, it indicates the name of the schema, the database or the physical model

Note: An element which is annotated {sqlName} must have a name composed of more than one character.

Annotating an operation

Tagged values on an operation

The tagged values used to annotate operations are presented in the following table:

The ... tagged value	Role	SQL
{tgInsert}	Indicates that the code for the operation containing an "*sqlCode*" note is a trigger set off on an "*insert*" operation.	(ON) INSERT
{tgUpdate}	Indicates that the code for the operation containing an "*sqlCode*" note is a trigger set off on an "*update*" operation.	(ON) UPDATE
{tgDelete}	Indicates that the code for the operation containing an "*sqlCode*" note is a trigger set off on an "*delete*" operation.	(ON) DELETE
{tgBefore}	Indicates that the trigger operation must be run before updating the database (Oracle).	BEFORE
{tgAfter}	Indicates that the trigger operation must be run after updating the database (Oracle).	AFTER

The {tgInsert} tagged value

At least one of the {tgInsert}, {tgUpdate} and {tgDelete} tagged values must be present on an operation stereotyped <<trigger>>. They determine when the trigger is launched.

The {tgUpdate} tagged value

In Oracle, the user can add as parameters the names of the columns which must be observed for update.

At least one of the {tgInsert}, {tgUpdate} and {tgDelete} tagged values must be present on an operation stereotyped <<trigger>>. They determine when the trigger is launched.

The {tgDelete} tagged value

At least one of the {tgInsert}, {tgUpdate} and {tgDelete} tagged values must be present on an operation stereotyped <<trigger>>. They determine when the trigger is launched.

The {tgBefore} tagged value

In Oracle, one and only one of the {tgBefore} and {tgAfter} tagged values must be defined on an operation stereotyped <<trigger>>.

The {tgAfter} tagged value

In Oracle, one and only one of the {tgBefore} and {tgAfter} tagged values must be defined on an operation stereotyped <<trigger>>.

Notes on an operation

The notes used to annotate operations are presented in the following table:

The ... note type	is used to...
sqlCode	indicate the SQL code of an operation, so as to allow the entry of the code for a stored procedure or a trigger.

Stereotypes values on an operation

The stereotypes used to annotate operations are presented in the following table:

The ... stereotype	is used to...
<<createView>>	indicate that the operation contains SQL code, placed on an "*sqlCode*" note, allowing a view to be created.
<<trigger>>	indicate that the operation is a trigger.
<<storedProcedure>>	indicate that the operation is a stored procedure.

The <<createView>> stereotype

The <<createView>> stereotype indicates that the operation contains SQL code, placed in an "sqlCode" note, which allows the creation of a view.

This stereotype must be placed on a class stereotyped <<view>>.

The <<trigger>> stereotype

The <<trigger>> stereotype indicates that the operation is a trigger. The SQL code of the trigger must be contained in an "sqlCode" note.

The operation must be annotated by at least one of the {tgInsert}, {tgUpdate} and {tgDelete} tagged values.

In Oracle, the operation must also have either the {tgBefore} or {tgAfter} tagged value present.

The <<storedProcedure>> stereotype

The <<storedProcedure>> stereotype indicates that the operation is a stored procedure and contains SQL code placed in an "sqlCode" note.

Annotating a package

Notes on a package

The notes used to annotate packages are presented in the following table:

The ... note type	is used to...
sqlBefore	define the SQL code which will be added at the beginning of the .SQL file associated with the modeled package.
sqlAfter	indicate the SQL code which will be added to the end of the .SQL file associated with the modeled package.

Stereotypes on a package

The stereotypes used to annotate packages are presented in the following table:

The ... stereotype	is used to...
<<schema>>	indicate that the package is a schema.
<<database>>	indicate that the package is a database.
<<sqlPhysicalModel>>	indicate that the package is a physical model.

The <<schema>> stereotype

The <<schema>> stereotype indicates that the package is a schema in which classes stereotyped <<table>> or <<view>> will be defined.

In a physical model, a <<schema>> can only be found on a package stereotyped <<sqlPhysicalModel>> or <<database>>.

The <<database>> stereotype

The <<database>> stereotype indicates that the package is a database.

A <<database>> can only be found on a package stereotyped <<sqlPhysicalModel>>.

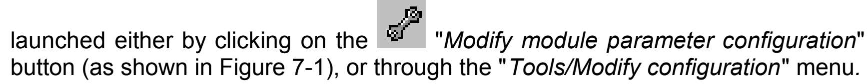
The <<sqlPhysicalModel>> stereotype

The <<sqlPhysicalModel>> stereotype indicates that the package is a physical model. It can contain packages stereotyped <<database>> or <<schema>>, or classes stereotyped <<table>>, <<view>> or <<procedureClass>>.

Chapter 7: Module configuration

Overview of module configuration

Introduction

The user can himself parameterize the *Objecteering/SQL Designer* module. This parameterization is carried out in the "*Modifying configuration*" dialog box, which is launched either by clicking on the  "*Modify module parameter configuration*" button (as shown in Figure 7-1), or through the "*Tools/Modify configuration*" menu.

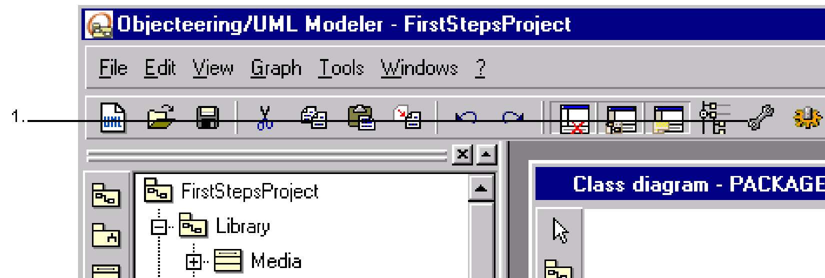


Figure 7-1. Launching the "*Modifying configuration*" dialog box

Module parameter sets

In the SQL parameter category, six different sets of parameters are available:

- General
- Physical generation
- Diagram generation on physical model
- SQL generation
- SQL generation filters
- External edition

General

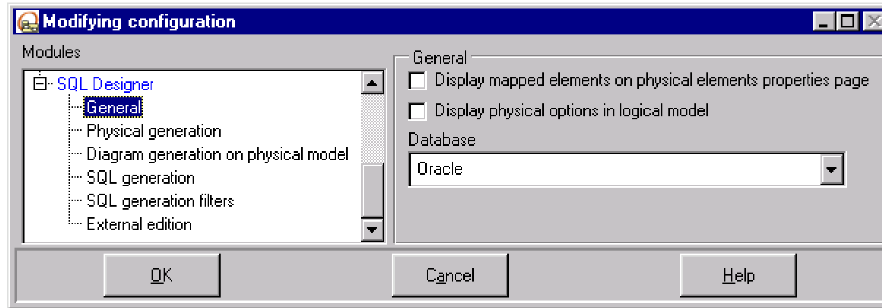


Figure 7-2. Editing the configuration of the *Objecteering/SQL Designer* module - the "*General*" sub-section

Configuration options are identical for *Oracle*, *Sybase* and *SQL Server*.

The ... option	is used to ...
Display mapped elements on physical elements properties page	display on a physical model element properties editor the element from which it was generated.
Display physical options in logical model	display on the properties editor, in a logical model, options that would normally only be displayed in the physical model.
Database	specify the target RDBMS.

Physical generation

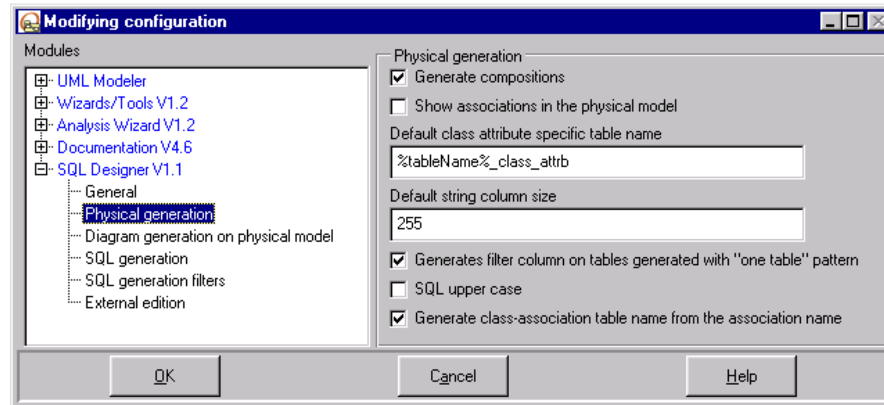


Figure 7-3. Editing the configuration of the *Objecteering/SQL Designer* module - the "*Physical generation*" sub-section

The ... option	is used to ...
Generate compositions	activate specific generation of compositions. If deactivated, compositions are handled as simple associations.
Show associations in the physical model	display associations in the physical model.
Default class attributes specific table name	determine the class attribute table name for each class which has attributes which are not annotated {globalTag}. %tableName% is replaced by the table name generated from the class.
Default string column size	determine the size of string columns where the "*Type constraint*" has not been defined on the source attribute.
Generates filter column on tables generated with "one table" pattern	generate a filter column on tables generated using the "one table" pattern.
SQL upper case	generate in upper case. This feature allows you to select the generation of all relational unit names in capital letters. If none is specified, the symbols correspond to those entered in Objecteering/UML.
Generate class-association table name from the association name	indicate that the name of the table generated from a class association is the name of the association. If the tickbox is not checked, the name of the table generated from a class association is the name of the class association.

Diagram generation on physical model

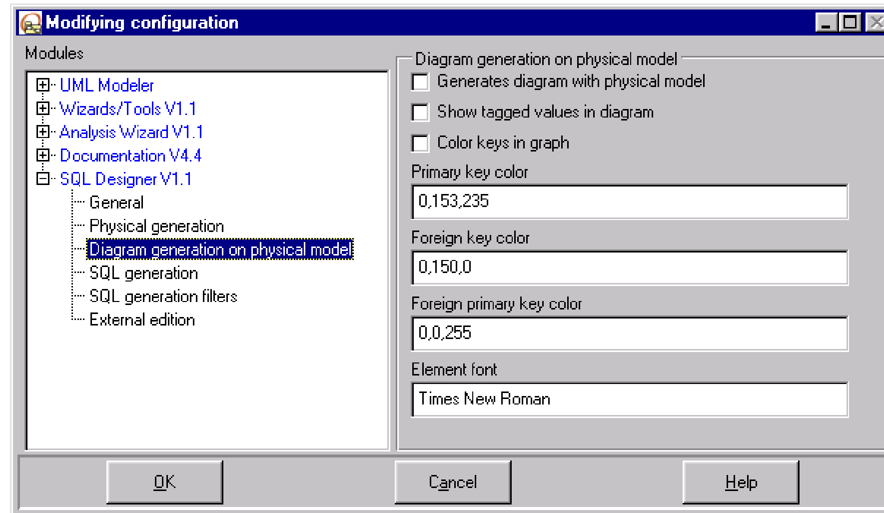


Figure 7-4. Editing the configuration of the *Objecteering/SQL Designer* module - the "*Diagram generation on physical model*" sub-section

The ... option	is used to ...
Generate diagram with physical model	automatically generate a class diagram on the physical model.
Show tagged values in diagram	display tagged values in the physical model class diagram.
Color keys in diagram	color attributes in the generated diagram to highlight primary and foreign keys.
Primary key color	determine the color of the primary key attribute. The color must have "red, green, blue" syntax, where red, green and blue are numbers included between 0 (for black) and 255 (full red, green and blue).
Foreign key color	determine the color of the foreign key attribute. The color must have "red, green, blue" syntax, where red, green and blue are numbers included between 0 (for black) and 255 (full red, green and blue).
Foreign primary key color	determine the color of the foreign primary key attribute. The color must have "red, green, blue" syntax, where red, green and blue are numbers included between 0 (for black) and 255 (full red, green and blue).
Element font	determine the font used in the diagram. Use the "Resources" button in the tool box and customize one of the element fonts, and then copy and paste the "font" field from the "Resources" dialog box to the module parameter.

SQL generation

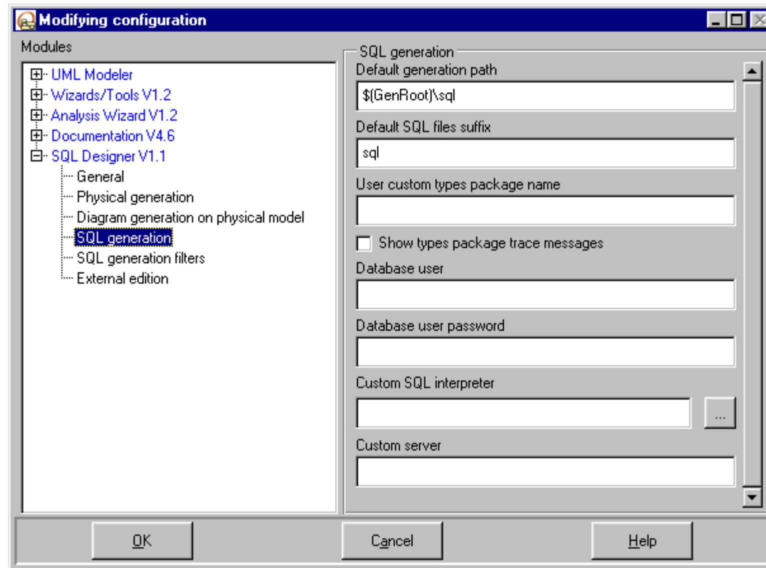


Figure 7-5. Editing the configuration of the *Objecteering/SQL Designer* module - the "*SQL generation*" sub-section

The ... option	is used to ...
Default generation path	specify the directory for the generation of SQL files.
Default SQL files suffix	specify the default suffix used for SQL files.
User custom types package name	specify the user customized type mapping project name. This parameter is optional and is left undefined by default.
Database user	indicate the name of the user of the database.
Database user password	indicate the password for the database user indicated.
Custom SQL interpretor	indicate the name and the path of the SQL interpretor.
Custom server	indicate the name and the path of the server.

SQL generation filters

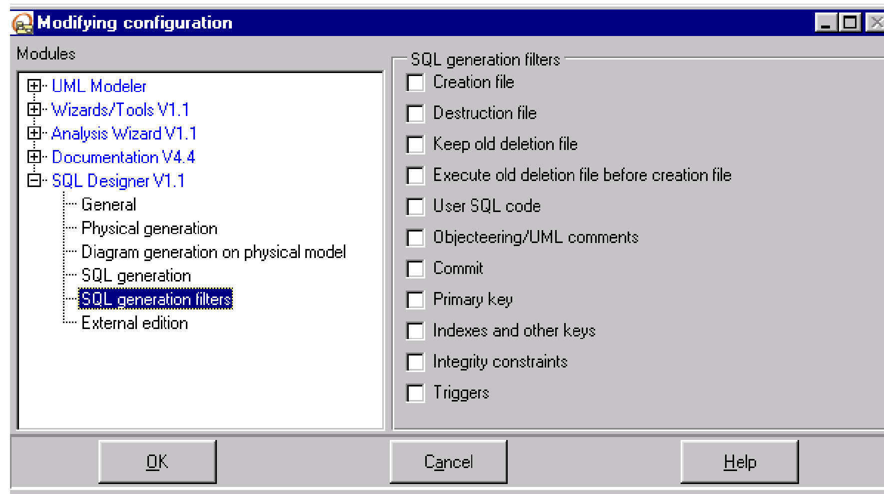


Figure 7-6. Editing the configuration of the *Objecteering/SQL Designer* module - the "*SQL generation filters*" sub-section

The ... option	is used to ...
Creation file	generate a creation file. This feature allows the generation of creation orders corresponding to the model's mapping.
Destruction file	generate a deletion file. This feature allows the generation of deletion orders corresponding to the model's mapping.
Keep old deletion file	keep a backup of the old delete file before each generation. It can be executed to cancel the old creation file.
Execute old deletion file before creation file	execute the backup delete file before the new creation file, which cancels the old creation file effect.
User SQL code	take into account the user SQL code.
Objecteering/UML comments	take into account Objecteering/UML comments. This is used to generate comments built by Objecteering/UML, in order to provide improved legibility of the SQL source produced.
Commit	generate "commit" orders. This generates a "commit" after each creation (or deletion) batch, in order to allow the RDBMS to record orders, without waiting for the interpretation of all the SQL orders to be completed.
SQL upper case	generate in upper case. This feature allows you to select the generation of all relational unit names in capital letters. If none is specified, the symbols correspond to those entered in Objecteering/UML.
Primary key	generate the primary key associated to the tables corresponding to the operation's model, if they are present.
Indexes and other keys	take into account secondary keys and indexes. This feature is used to generate the secondary keys and indexes associated to the tables corresponding to the operation's model, if they are present.
Integrity constraints	generate constraints deduced from associations. This feature selects those integrity constraints associated with the operation's units, if they are present.
Triggers	generate triggers.

External edition

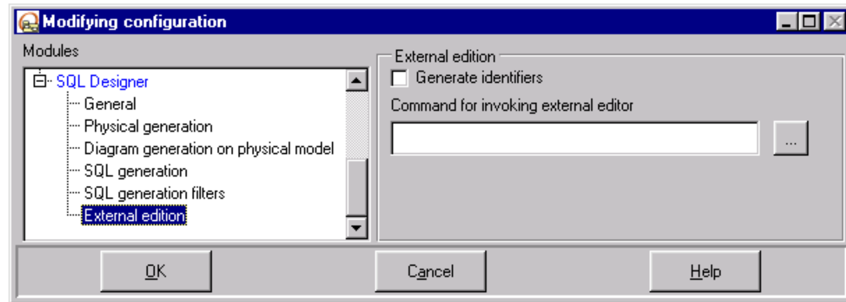


Figure 7-7. Editing the configuration of the *Objecteering/SQL Designer* module - the "*External edition*" sub-section

The ... option	is used to ...
Generate identifiers	This parameter has no use in *Objecteering/SQL Designer*.
External editor invocation command	This parameter has no use in *Objecteering/SQL Designer*.

Parameterizing by redefining J methods in a profile

To use this kind of parameterization, a license for the *Objecteering/UML Profile Builder* module is required.

Parameterizing the package unit

Parameterization

A parameterization method is used to define the default name used for the database.

Method

The signature of the parameterization method is as follows:

```
Schema:SetDefaultDbName() return String
```

Default

Its value is located by default in the following UML profile:

```
default#external#Code#RDB

SetDefaultDbName ()
   return String
{
   return "Default";
}
```

Parameterizing the class unit

Parameterization methods

The ... method	defines ...
SetTableName	the name of the table corresponding to the association
SetPrimaryKeyName	the name of the primary key declared in BD

Signatures

The signatures of the parameterization methods are as follows:

```
SetTableName () return String
SetPrimaryKeyName () return String
```

SetTableName method

```
// Default value
default#external#Code#RDB
   SetTableName () return String
{
   return Name;
}
```

SetPrimaryKeyName method

```
// Default value
default#external#Code#RDB
   SetPrimaryKeyName ()
      return String
{
      String ret = SetTableName();
      return.concat("_PK");
      return ret;
}
```

Parameterizing the association unit

Parameterization methods

The ... method	defines ...
SetTableName	the name of the table corresponding to the association
SetPrimaryKeyName	the name of the primary key declared in BD
SetColumnPrefix	the prefixes of the column names. These are added before the name of the corresponding attribute.

Signatures

The signatures of the parameterization methods are as follows:

```
AssociationEnd:SetTableName(
   ComposedClassName : in String ) return String

AssociationEnd:SetPrimaryKeyName(
   ComposedClassName : in String ) return String

AssociationEnd:SetColumnPrefix(
   OriginPrefix : inout String,
   DestPrefix : inout String ) return String
```

UML profile

These methods are declared in the following UML profile:

```
default#external#Code#RDB
```

SetTableName method

```
// Default value
AssociationEnd: default#external#Code#RDB#SetTableName
   (in String ComposedClassName)
    return String
{
   ret.concat(destR, "_of_", ComposedClassName);
}
```

SetPrimaryKeyName method

```
// Default value
AssociationEnd: default#external#Code#RDB#SetPrimaryKeyName
   (in String ComposedClassName)
    return String
{
   ret.concat("_PK");
   return ret;
}
```

SetColumnPrefix method

```
// Default value
AssociationEnd: default#external#Code#RDB#SetColumnPrefix
   (inout String OriginPrefix,
    inout String DestPrefix)
{
OriginPrefix = destRole() ; OriginPrefix.concat("_") ;
DestPrefix = originRole() ; DestPrefix.concat("_") ;
}
```

The *OriginPrefix* and *DestPrefix* strings are used as prefixes to the columns deduced from associations. Their purpose is to avoid repetition of column names within the same table, especially in the case of included associations, reflexive associations, or even in the case of classes with an association towards attributes of the same name.

Parameterizing attribute units

Parameterization methods

The ... method	defines ...
SetColumnName	the name of the column corresponding to the attribute.
SetIsNullVarName	the name of the attribute generated when the attribute may be null in the database (the case of the @not_null non specified attributes).

Note: The "*Type conversion*" paragraph explains how to use methods to parameterize types.

J method

The signatures of the parameterization methods are as follows:

```
Attribute:SetColumnName() return String

Attribute:SetIsNullVarName() return String
```

UML profile

These methods are declared in the following UML profile:

```
default#external#Code#RDB
```

SetColumnName method

```
// Default value
SetColumnName ()
      return String
{
      return Name;
}
```

SetIsNullVarName method

```
Default value
SetIsNullVarName ()
   return String
{
ret.concat("IsNull");
return ret;
} //
```

Parameterizing by adding notes predefined by the generation

Free SQL sections: sqlAfter and sqlBefore

The user has the possibility of defining notes such as *sqlafter* or *sqlbefore* within packages or classes, according to whether he wishes to insert SQL at the beginning or at the end of generation on the current unit.

This feature is especially useful for specifying in pure SQL, at invariant and pre- and post-condition level, integrity constraints which are not deduced from the model.

Example

```
text : sqlAfter
insert into graphic_object values (10,11);
insert into graphic_object values (50,100);
grant select on table graphic_object to public;
```

Notes defined on a package

The ... note type	is inserted ...
sqlBefore	at the beginning of the file containing the SQL instructions generated for the package.
sqlAfter	at the end of the file containing the SQL instructions generated for the package.

Notes defined on a class

The ... note type	is inserted ...
sqlBefore	at the beginning of the SQL instructions related to the class
sqlAfter	at the end of the SQL instructions related to the class

Note: These text zones can be created through *Objecteering/UML Profile Builder* in order to automate the generation of model context information.

Chapter 8: Oracle Annex

Introduction

Overview

The object of this annex is to present the method used to adapt generation to the Oracle V8 RDBMS. The annex is divided into the following model units: class, association and attribute. For each unit, explanations are related to generation carried out and optional annotations. The last section deals with the characteristics of the application part.

Generation deals especially with the mapping of constraints and types.

Annotations specific to Oracle allow RDBMS features to be used in a simple way (*cluster, optimization, additional constraints*).

Environment variables

To be able to run SQL scripts to create tables, a user name and a password must exist. Objecteering/UML uses the *O_SQL_USER* environment variable that indicates them: *name/password/serverName*.

Class - Generation specific to Oracle

Primary key

The primary key is directly implanted after use of the instructions of the SQL89 norm: *CONSTRAINT ... PRIMARY KEY*.

For example:

```
ALTER TABLE class ADD (
    CONSTRAINT class_PK PRIMARY KEY (attribute1, attribute2)
) ;
```

Class annotations

Tagged values

The tagged values available at class level are presented in the following table:

The ... tagged value	is used to ...	Oracle
{key}	define "*unique*" constraints.	UNIQUE
{check}	define a "*check*" constraint.	CHECK
{cluster}	link the table to a predefined cluster.	CLUSTER
{initrans}	change the characteristics of the table.	INITTRANS
{maxtrans}	as above.	MAXTRANS
{pctfree}	as above.	PCTFREE
{pctused}	as above.	PCTUSED
{storage}	as above.	STORAGE
{noDDL}	delete the generation of the definition language.	
{tablespace}	specify the logical storage unit of the data.	TABLESPACE
{indexSpace}	specify the tablespace for the index.	TABLESPACE
{indexStorage}	specify the STORAGE class for creating the index.	STORAGE

The {check} tagged value

This tagged value's unique parameter is the logical condition, used as is, for building the clause's SQL. We recommend that you place this parameter between inverted commas for the syntactical analyzer.

Example:

```
{check}("(att1 + att2)between 1 and 12")
```

The {cluster} tagged value

This tagged value is used to link the table to a pre-existing cluster (for example created within a "*sqlbefore*" item).

The descriptor of the tagged value gives the cluster's name, whereas the parameters correspond to the list of columns associated to the cluster.

The {key} tagged value

This *{key}* tagged value allows the definition of uniqueness constraints associated with the generated table. The descriptor gives the name, whilst the parameters provide the necessary columns.

Example:

```
ALTER TABLE <table name> ADD CONSTRAINT (<constraint name>
UNIQUE (<attribute name>))
```

The {initrans}, {maxtrans}, {pctfree}, {pctused} and {storage} tagged values

These tagged values have as unique parameter the value of the Oracle parameter, with the same name as will be generated in this case. They must not be used together with a cluster (please see *Oracle SQL 4-210*).

The {tablespace} tagged value

This tagged value is used to specify the logical storage unit of the data. Classes not annotated with *{tablespace}* are stored in the *SYSTEM tablespace*.

The {indexSpace} tagged value

This tagged value allows the specification of the "*tablespace*" where the class index will be created.

The {indexStorage} tagged value

This tagged value is used to specify the "*STORAGE*" Oracle clause for indexes.

The parameter of this tagged value must contain, between inverted commas, the different Oracle parameters associated to a parameter's value.

Example:

```
"INITIAL 1024 NEXT 2048"
```

Association - Generation specific to Oracle

Referential constraints

Referential constraints are mapped by the following instructions:

```
SQL89 CONSTRAINT ... FOREIGN KEY ... REFERENCES.
```

Parameterization

Referential constraint names can be parameterized using the *SetForeignKeysNames* J method which simply has to be redefined under a UML Profile. The method's parameters are as follows:

The ... parameter	represents ...
ComposedClassName	the name of the composed class
fkDest	the destination "foreign key"
fkOrigin	the origin "foreign key"

SetForeign-KeysNames method

```
// Default value for Oracle.
AssociationEnd:default#external#Code#RDB#DDL#Oracle#
   SetForeignKeysNames
      (in String ComposedClassName,
       inout String fkOrigin,
       inout String fkDest)
{

   String fkO;
   String fkD;

   fkO = SetTableName(ComposedClassName);
   fkD = SetTableName(ComposedClassName);
   fkO.concat("_OFK");
   fkD.concat("_DFK");
   fkOrigin = fkO;
   fkDest = fkD;

}
// method SetForeignKeysNames
```

Multiplicity constraints

The generation of tables for Oracle V8 uses triggers to map multiplicity constraints in associations.

Multiplicity constraints are checked after insertion by a trigger named TI_name of the table. However, Oracle cannot be used to generate an *AFTER DELETE* type trigger, able to manage minimum multiplicities during a deletion.

We will generate a multiplicity test through association orientation if the association is mutual with m-n multiplicity.

Example of multiplicity constraints

```
DDL generated for an association with the 0-5 multiplicity:
CREATE TRIGGER TI_role2_of_class
    BEFORE INSERT
    ON role2_of_class
    FOR EACH ROW
    DECLARE
        dummy INTEGER;
    BEGIN
        SELECT COUNT(*)
            INTO dummy
            FROM role2_of_class
            WHERE clef = :new.clef;
        IF NOT(dummy < 5)
            THEN raise_application_error(-20501,
                      'role2_of_class : May not insert
element');
        END IF;
    END ;
/
```

Association annotations

Tagged values

The tagged values available at association level are presented in the following table.

The ... tagged value	allows ...	Oracle
{check}	as for class	CHECK
{cluster}	as for class	CLUSTER
{initrans}	as for class	INITRANS
{maxtrans}	as for class	MAXTRANS
{onDeleteCascade}	the addition of the "ON DELETE CASCADE" option to the constraints deduced from the model's associations	ON DELETE CASCADE
{pctfree}	as for class	PCTFREE
{pctused}	as for class	PCTUSED
{storage}	as for class	STORAGE
{tablespace}	as for class	TABLESPACE

It is necessary to specify the tagged values other than {*onDeleteCascade*} on each orientation of the association in the case of a mutual association.

{onDeleteCascade}

This tagged value offers the possibility of deleting foreign keys which reference a primary key when the primary key is deleted.

Attribute - Generation specific to Oracle

Mapping types

The correspondence between the types of the different "*spaces*" is presented in the table below.

Model	ORACLE Data Type
integer	INTEGER
{short integer}	INTEGER
{long integer}	INTEGER
{unsigned integer}	INTEGER
{boolean}	INTEGER
{real}	FLOAT
{long real}	FLOAT
string(n)	VARCHAR2(n)
char	CHAR (1)
set {array} (n) of char	VARCHAR2 (n)

Note: The types called Oracle Data Type here correspond to the internal type described in the Programmer's Guide to the Oracle Precompilers.

Attribute annotations

Tagged values

The tagged values available at attribute level are presented in the following table:

The ... tagged value	is used to ...	Oracle
{check}	declare a "check" type constraint in the column associated to the attribute in question.	CHECK
{default}	give the default value in the column	DEFAULT
{null}	specify that the column is NULL type	NULL
{unique}	declare that the column as unique	UNIQUE

The {check} tagged value

The {check} tagged value allows the specification of a Boolean expression in a column, which will have to be checked so that a tuple is inserted in the table.

See the {check} tagged value on classes for further details.

The {null} tagged value

This tagged value imposes the generation of a "*NULL*" constraint in the column corresponding to the attribute.

This constraint allows the use of an "*undetermined*" when inserting a tuple, for which data has not been entered in the column.

Distributed databases

Overview

During a transaction, it is possible to reference several remote tables, in addition to local tables. Generation takes into account these functions, by adding the name of the feature to the handling language, after the table's complete name. This notation interpreted by Oracle allows "*transparent*" access to distant data.

Tagged values

The ... tagged value	is used to ...	Oracle
{dbName}	specify the name of the service and the name of the facade package.	@

Example: Annotation of the "*Employee*" class with the *{dbname}*(*hq,objecteering, com*) tagged value. A request in the relational table resulting from the "*Employee*" class will take the following form:

```
SELECT * FROM Employee@hq.objecteering.com;
```

Consequences for generation

The definition language related to the classes annotated with the *{dbName}* tagged value is separate from the rest of the code generated for local classes. A file is generated for each annotated class. The file name is the concatenation of the class name, with the "*Of*" character string, and with the first parameter passed to the *{dbName}* tagged value.

Example of file name: *EmployeeOfhq*.

Restrictions

The generation checks that there is no association between classes annotated with a different database name.

Remarks

To reach a distant database, the service named (DNS) must be operational. If this is not the case, a link towards the distant database must be created in the following way:

```
CREATE DATABASE LINK hq.objecteering.com USING
'service_name';
```

The service name should correspond to the data entered in the "*tnsnames.ora*" file. (please refer to the SQL*Net Administrator's guide).

Parameterizing generation - Specificity

Overview

Oracle allows the use of specific types, such as *LONG, RAW, LONG RAW, LONG VARCHAR*, etc. Furthermore, *Objecteering/SQL Designer* has attribute parameterization, which allows you to use these types.

Available types

The available types are as follows:

```
number, long, date, longraw, raw, rowid, clob, blob, bfile
and nclob.
```

Parameterization

Parameters can be provided by defining the "*Type constraint*" field for the attribute. These specific types are parameterized using the {type} (<specific type>) tagged value set on an attribute.

Example

```
{type}(number), {type}(long), ...
```

Chapter 9: Sybase Annex

Introduction

Overview

The object of this annex is to present the method used to adapt generation to the *Sybase V11 RDBMS*. The annex is divided into the following model units :class, association and attribute. For each unit, explanations are related to the generation carried out and the optional annotations. The last section deals with the characteristics of the application part.

The generation deals especially with the mapping of constraints and types.

Annotations specific to Sybase allow RDBMS features to be used in a simple way (optimization, additional constraints).

Environment variables

The compilation of applications for the *Sybase RDBMS* requires the presence of the following environment variables:

- SYBASE must contain the *Sybase root directory path*
- PATH must contain *$SYBASE/bin*
- LANG must not be positioned to us

The execution of applications produced for the *Sybase RDBMS* requires the presence of the following environment variables:

- *O_SQL_USER* must contain the name of the *Sybase RDBMS* known password, which will be used to create relational tables. The format of this variable is as follows: *name/passwd*.
- LANG must not be positioned to us.

End of the SQL command

Each Transact-SQL statement must end with the "*go*" command (this can be renamed). This command replaces the traditional ';' in SQL.

Identifiers

The size of *Sybase* identifiers is limited to 30 characters. The first character should be alphabetical.

Case sensitivity

Sybase standard installation makes the SQL server "*case sensitive*". This has the effect of requiring that SQL generation conform with the case of the UML model classes or attributes. The "*SQL upper case*" parameter must not be selected in this case.

However, it is possible to make identifiers case-sensitive, by reconfiguring the "*sort*" order" of the SQL server (see Sybase SQL Server, Ref Manual Vol 1 page 5-41).

You will then be free to specify the desired case option when generating the SQL.

Package annotations

Tagged values

The tagged values specific to Sybase at package level are presented in the following table:

The ... tagged value	T	is used to ...	Sybase
genCascade	S	take into account the {onDeleteCascade} tagged value in SQL generation	CREATE TRIGGER ...

The {genCascade} tagged value

See the {onDeleteCascade} tagged value in the "*Association annotations*" section.

The {schemaBD} tagged value

The {schemaBD} tagged value has no impact on the *Sybase* coupling.

Class - Generation specific to Sybase

Primary key

The primary key (tagged value *{primaryKey}*) is directly inserted after use of instructions of the *SQL89* norm: *CONSTRAINT* ... *PRIMARY KEY*.

For example:

```
  ALTER TABLE class ADD
     CONSTRAINT class_PK PRIMARY KEY (attribute1,
attribute2)
```

Note: Only one primary key can be specified by the *ALTER TABLE* command.

Its deletion (drop) is carried out using the *DROP CONSTRAINT* clause of the *ALTER TABLE* command.

For example:

```
  ALTER TABLE class
     DROP CONSTRAINT class_PK
```

Note: Only one primary key can be deleted by the *ALTER TABLE* command.

Class annotations

Tagged values

The tagged values specific to *Sybase* possible at class level are presented in the following table:

The ... tagged value	is used to ...	Sybase
{primaryKey}	define the primary key of the class and create an index with direct access.	PRIMARY KEY
{key}	declare a key on a combination of attributes.	UNIQUE
{index}	specify an index on a combination of attributes.	CREATE INDEX
{check}	declare a "check" type constraint on a combination of attributes.	CHECK
{fillfactor}	specify the percentage of pages filled by the index	WITH FILLFACTOR
{tablespace}	specify the name of the segment in which the table will be allocated.	ON
{partition}	specify the number of pages chained together in the table.	PARTITION
{maxRowsPerPage}	specify the maximum number of tuples per page	WITH MAX_ROWS_PER_PAGE

The {key} tagged value

This *{key}* tagged value is used to define uniqueness constraints associated with the generated table. The descriptor gives the name, whilst the parameters provide the necessary columns.

The {check} tagged value

The *{check}* tagged value allows the specification of a *Boolean* expression in a combination of columns of the table, that will have to be checked so that a tuple be inserted in the table.

This tagged value has for unique parameter the logical condition, used as is, for building the clause's SQL. We recommend that you place this parameter between inverted commas for the syntactical analyzer.

Example:

```
{check}("(att1 + att2) between 1 and 12")
```

Note: Remember in this case that if you wish to incorporate the " character in a tagged value's parameter, it is necessary to enter the ~ character before it.

The {fillfactor} tagged value

This tagged value specifies the fill factor of each index page.

The *{fillfactor}* must be between 1 and 100.

If the *{fillfactor}* is not specified, the default value is 0.

A 0 *{fillfactor}* creates indexes clustered with full pages and indexes not clustered by full page sheets.

Note: (please refer to Sybase SQL Server Reference Manual Vol 1 pages 3-79).

The {partition} tagged value

This tagged value specifies the number of chain pages in the table. It allows the SQL to execute insertions at the same time on the last page of each chain.

The partition must be greater than or equal to 2.

By default, there is only one chain of pages per table.

Note: (please refer to Sybase SQL Server Reference Manual Vol 1 pages 3-15).

The {maxRowsPerPage} tagged value

This tagged value allows the specification of the maximum number of tuples per page in the table.

{maxRowsPerPage} must be between 0 and 256.

If the *{maxRowsPerPage}* is not specified, the default value is 0.

A 0 value for *{maxRowsPerPage}* creates indexes that are clustered with full pages and indexes not clustered by full page sheets.

Note: (please refer to Sybase SQL Server Reference Manual Vol 1 pages 3-80).

The {tablespace} tagged value

This tagged value specifies a specific segment of allocation of the table's pages.

The segment will have to have been created first by the administrator using the *sp_addsegment* procedure.

The {schemaBD} tagged value

The *{schemaBD}* tagged value has no impact on the *Sybase* coupling.

Association - Generation specific to Sybase

Referential constraints

Referential constraints are mapped by SQL instructions:

```
FOREIGN KEY ... REFERENCES.
```

Note: only one referential constraint can be specified using the Transact SQL "alter table" command.

Example of referential constraint

SQL generated to specify the referential constraints of an Association:

```
ALTER TABLE role_of_table1
   ADD FOREIGN KEY (att1_of_role_of_table1)
     REFERENCES table1 (att1)

ALTER TABLE role_of_table1
   ADD FOREIGN KEY (att2_of_role_of_table1)
    REFERENCES table2 (att2)
```

Parameterization

Referential constraint names can be parameterized using the *SetForeignKeysNames* J method that simply has to be redefined under a UML profile. The method's parameters are as follows:

The ...parameter	**represents ...**
ComposedClassName	the name of the composed class
fkDest	the destination "*foreign key*"
fkOrigin	the origin "*foreign key*"

SetForeignKeysNames method

```
// Default value for Sybase
AssociationEnd:default#external#Code#RDB#DDL#Sybase#
   SetForeignKeysNames
      (in String ComposedClassName,
       inout String fkOrigin,
       inout String fkDest)
{

   String fkO;
   String fkD;

   fkO = SetTableName(ComposedClassName);
   fkD = SetTableName(ComposedClassName);
   fkO.concat("_OFK");
   fkD.concat("_DFK");
   fkOrigin = fkO;
   fkDest = fkD;

}
// method SetForeignKeysNames
```

Multiplicity constraints

The generation of tables for Sybase V11 uses triggers to map multiplicity constraints in associations.

Multiplicity constraints are checked after insertion by a trigger named TI_*name of the table* and after deletion by a trigger named TD_*name of the table*.

We will generate a multiplicity test through association navigation if the association is mutual with the m-n multiplicity.

Example of multiplicity constraint

SQL generated for an association with 0-5 multiplicity:

```
CREATE TRIGGER TI_role2_of_class
ON role2_of_class
FOR INSERT AS
IF NOT((SELECT COUNT(*)
      FROM role2_of_class, inserted
       WHERE role2_of_class.key = inserted.clef) <= 5)
   BEGIN
      ROLLBACK TRANSACTION
      RAISERROR 99999 "role2_of_class : May not insert
element,
cardinality constraint violation"
   END
```

Note: In this case, there is no generation of the FOR DELETE trigger to check minimum multiplicity, as it is 0.

{onDeleteCascade}

SQL generated for a tuple deletion in the table C1 or C2.

Let us suppose that the role_of_C1 is the table which implements an association between C1 and C2, when a tuple of C1 or C2 is deleted, and that we also want to delete from the table which makes up the association between C1 and C2 all tuples which have the key of the deleted tuple as foreign key.

We will generate a trigger for each class concerned by the association:

```
CREATE TRIGGER TDCAS_C1
    ON C1
    FOR DELETE AS
    DELETE role_of_C1
    FROM role_of_C1, deleted
    WHERE role_of_C1.clefC1 = deleted.clefC1
```

and

```
CREATE TRIGGER TDCAS_C2
    ON C2
    FOR DELETE AS
    DELETE role_of_C1
    FROM role_of_C1, deleted
    WHERE role_of_C1.clefC2 = deleted.clefC2
```

Association annotations

Tagged values

The tagged values specific to *Sybase* and which are available at association level are presented in the following table:

The ... tagged value	**is used to ...**	**Sybase**
{fillfactor}	as for class	WITH FILLFACTOR
{tablespace}	as for class	ON
{partition}	as for class	PARTITION
{maxRowsPerPage}	as for class	WITH MAX_ROWS_PER_PAGE
{onDeleteCascade}	maintain referential integrity constraints on associations.	CREATE TRIGGER ...

In the case of a mutual association, it is necessary to specify the tagged values other than *{onDeleteCascade}* on each orientation of the association.

{onDeleteCascade}

This tagged value offers the possibility of deleting the foreign keys that reference a primary key when this key is deleted.

Indeed, when we delete a tuple with a primary key, we also want to delete tuples with a corresponding foreign key in the dependent tables. This is especially true if the table is concerned by an association.

As far as the implementation of this tagged value is concerned, Sybase does not provide the *ON DELETE CASCADE* (standard SQL92) clause on the referential integrity constraints.

Furthermore, it is not possible to use the triggers which come with the referential integrity constraints in a simple way, since these are checked *before* the launching of the triggers.

The *Sybase* coupling suggests two ways of interpreting the *{onDeleteCascade}* tagged value:

- The first is to ignore this tagged value and to let the application delete associations between instances before deleting the instances themselves. This option is chosen by default.
- The second is to implement this property in the SQL, using the triggers. In this case, you must also simulate the referential integrity constraints which use triggers on the insertion.
- In relation to this, the *{gen cascade}* tagged value indicates that the *{onDeleteCascade}* tagged value should provoke the generation of the triggers necessary in the SQL.
- The drawback in this is that it makes the *FOREIGN KEY* referential integrity constraints disappear from the produced SQL. *REFERENCES* ordinarily used on tables which materialize associations.
- This choice should be avoided if you wish to reverse engineer the produced SQL, as the associations between the tables no longer clearly appear.

Attribute - Generation specific to Sybase

Mapping types

The correspondence between the types of the different "*spaces*" is presented in the table below:

Model	Sybase Data Type
integer	INTEGER
{short integer}	SMALLINT
{long integer}	INTEGER
{unsigned integer}	INTEGER
boolean	BIT
real	REAL
{long real}	FLOAT (p)
string(n)	VARCHAR (n)
char	CHAR (1)
{unsigned char}	TINYINT
set {array}(n) of char	CHAR (n)

Note 1: Types called here under Sybase Data Type correspond to the SQL Server Datatype, described in the *Open Client Embedded SQL/C Programmer's Manual*, pages 4-16 and 4-17.

Note 2: The SQL VARCHAR type is a character type in which the blanks at the end have been removed.

Note 3: The Objecteering/UML string(*) and string() types do not have an equivalent type in Sybase. Indeed, it is compulsory to specify the size of a string, the size of the VARCHAR of Sybase having been limited to 255 characters.

Attribute annotations

Tagged values

The tagged values specific to Sybase available at attribute level are presented in the following table:

The ... tagged value	is used to ...	Sybase
{default}	give the default value in the column	DEFAULT
{check}	declare a "check" type constraint in the column associated to the attribute in question.	CHECK
{null}	specify that the column is NULL type	NULL
{unique}	declare the column as unique	UNIQUE

The {check} tagged value

The *{check}* tagged value allows the specification of a Boolean expression *in a column*, that will have to be checked in order that a tuple be inserted in the table.

The name of the column in question must appear in the expression and must be the only one.

See the {check} tagged value on the classes for further details.

The {null} tagged value

This tagged value imposes the generation of a "*NULL*" constraint in the column corresponding to the attribute.

This constraint allows the use of an "*undetermined*" when inserting a tuple for which data has not been entered in the column.

The {unique} tagged value

This tagged value is used to specify that all the values of the column are unique.

Specific SQL errors

SQL errors

The SQL generated for Sybase RDB can, in certain cases, provoke errors which are then transferred to the application.

These errors are as follows:

- 20501 *Tablename* : May not insert element, cardinality constraint violation

This error is launched when the upper limit of the association's multiplicity is surpassed. In this case, a rollback of the transaction is automatically launched.

- 20502 *Tablename* : May not delete element, cardinality constraint violation

This error is launched when the lower limit of the association's multiplicity is surpassed. In this case, a rollback of the transaction is automatically launched.

- 20503 *Tablename* : May not insert element, primary key of TableName unknown

This error is triggered when you try to create an association towards an instance that is not yet in the database. This error can only be launched if the {gen_cascade} tagged value is present on the association's package. Otherwise, the FOREIGN KEY REFERENCES referential integrity constraint will be violated.

- 20504 *Tablename* : May not update element

This error is set to prevent you from updating tables which materialize associations. Indeed, in these tables you may only insert or delete tuples, but you may never update them. This error can only be launched through external access to the generated code by coupling the Sybase RDB.

Parameterizing generation - Specificity

Overview

Sybase allows the use of specific types, such as *NUMERIC, DATETIME, IMAGE*, etc. Furthermore, *Objecteering/SQL Designer* possesses attribute parameterization which allows you to use these types.

Available types

The types available are as follows:

```
numeric, decimal, double,  smallmoney, money, smalldatetime,
datetime, text, binary, varbinary and image
```

Parameterization

Parameters can be provided by defining the "*Type constraint*" field of the attribute.

These specific types are parameterized using the {type} (<specific type>) tagged value set on an attribute.

Example

```
{type}(numeric), {type}(money), ...
```

Restrictions

Generated SQL upper or lower case

A bug in the Sybase Transact SQL interpreter, which does not recognize the predefined "*inserted*" and "*deleted*" tables when they appear in upper case in the SQL, makes it impossible to position the "*SQL Upper Case*" parameter.

Chapter 10: SQL Server Annex

Introduction

Overview

The object of this annex is to present the method used to adapt generation to the *SQL Server RDBMS*. The necessary information is included in chapter 9 of this user guide, and for each section, the user is referred to the relevant section of chapter 9.

The annex is divided into the following model units: class, association and attribute. For each unit, explanations are related to the generation carried out and the optional annotations. The last section deals with the characteristics of the application part.

The generation deals especially with the mapping of constraints and types.

Annotations specific to SQL Server allow RDBMS features to be used in a simple way (optimization, additional constraints).

Package annotations

For information on package annotations, please refer to the "*Package annotations*" section of chapter 9 of this user guide.

Class - Generation specific to SQL Server

For information on the generation specific to SQL Server concerning classes, please refer to the "*Class - Generation specific to Sybase* " section of chapter 9 of this user guide.

Class annotations

For information on class annotations, please refer to the "*Class annotations*" section of chapter 9 of this user guide.

Association - Generation specific to SQL Server

For information on the generation specific to SQL Server concerning classes, please refer to the "*Association - Generation specific to Sybase*" section of chapter 9 of this user guide.

Association annotations

For information on association annotations, please refer to the "*Association annotations*" section of chapter 9 of this user guide.

Attribute - Generation specific to SQL Server

For information on the generation specific to SQL Server concerning attributes, please refer to the "*Attribute - Generation specific to SQL server*" section of chapter 9 of this user guide.

Attribute annotations

For information on attribute annotations, please refer to the "*Attribute annotations*" section of chapter 9 of this user guide.

Specific SQL errors

For information on specific SQL errors, please refer to the "*Specific SQL errors*" section of chapter 9 of this user guide.

Parameterizing generation - Specificity

For information on the specificity of generation parameterization, please refer to the "*Parameterizing generation - Specificity*" section of chapter 9 of this user guide.

Restrictions

For information on restrictions, please refer to the "*Restrictions*" section of chapter 9 of this user guide.

Chapter 11: Calling on-line module commands

Calling on-line commands

Syntax

An on-line command is called using an instruction, as shown below:

```
objingcl -db <base_name>
-prj <project_name>
-mdl SQLModule
-cmd <command_Name>
<metaclass>::*::<object_name>
```

Commands which can be invoked

The ... command	on the ... metaclass	runs ...
generateDDL	DDLProduct	DDL code generation
editDDL	DDLProduct	Visualization
execDDL	DDLProduct	Execution

For example:

```
objingcl db DemoSQL -prj testRDB -mdl SQLModule -cmd
generateDDL
DDLproduct::*::basicRDB_DDL
```

Index

"Not null" attributes 4-53
"sqlAfter" note 6-7, 6-28, 6-39, 7-21
"sqlBefore" note 6-7, 6-28, 6-39, 7-21
"sqlCode" note 5-16, 6-7, 6-32, 6-34, 6-37
- association pattern 4-39
{alterTable} tagged value 6-4, 6-20
{check} tagged value 8-5
{classAttributesTableName} tagged value 4-19, 6-5, 6-25
{cluster} tagged value 6-3, 6-13, 6-23, 8-5
{compositeKey} tagged value 6-4, 6-20
{dbName} tagged value 6-6, 6-29, 8-14
  Consequences for generation 8-14
{external} tagged value 4-26, 5-15, 6-3, 6-13
{fillfactor} tagged value 6-3, 6-14, 6-23
{foreignKey} tagged value 4-25, 6-4, 6-19
{gen_cascade} tagged value 9-15
{globalTable} tagged value 4-17, 6-4, 6-19
{index} tagged value 4-3
{indexKey} tagged value 4-57, 6-4, 6-19
{indexName} tagged value 4-57, 6-5, 6-27
{indexSpace} tagged value 6-5, 6-24, 8-5
{indexStorage} tagged value 6-5, 6-25, 8-5
{initrans} tagged value 6-3, 6-13, 6-23, 8-5
{key} tagged value 8-5
{keyName} tagged value 4-57, 6-5, 6-26
{maxRowsPerPage} tagged value 6-3, 6-14, 6-23
{maxtrans} tagged value 6-3, 6-14, 6-24, 8-5
{noDDL} tagged value 6-5, 8-5
{not_null} tagged value 4-26
{oid} tagged value 6-5, 6-26
{onDeleteCascade} tagged value 4-48, 6-16, 9-5, 9-13
{onDeleteCascase} tagged value 6-3
{oneTable} tagged value 4-8, 5-11, 6-5, 6-25
{oneTablePerClass} tagged value 4-8, 5-11, 6-5, 6-25
{oneTablePerConcreteClass} tagged value 4-8, 5-11, 6-5, 6-25
{partition} tagged value 6-3, 6-13, 6-26
{pctfree} tagged value 6-3, 6-13, 6-24, 8-5
{pctused} tagged value 6-3, 6-13, 6-24, 8-5
{persistence} tagged value 4-7, 4-24, 4-40, 4-55, 5-11, 6-4, 6-20, 6-27, 6-33
{primaryKey} tagged value 4-3, 4-21, 4-24, 4-37, 4-54, 5-15, 6-3, 6-16, 6-19
{schemaBD} tagged value 9-5
{sqlDefault} tagged value 4-20, 5-13, 6-4, 6-19
{sqlDoNotUpdate} tagged value 4-37, 5-9, 5-19

{sqlName} tagged value 4-26, 5-7, 6-6, 6-34
{sqlOptimized} tagged value 4-40, 5-15, 6-3, 6-16
{sqlType} tagged value 4-20, 4-33, 5-13, 6-4, 6-20
{storage} tagged value 6-3, 6-14, 6-24, 8-5
{tablespace} tagged value 6-3, 6-13, 6-26, 8-5
{tgAfter} tagged value 6-6, 6-36
{tgBefore} tagged value 6-6, 6-36
{tgDelete} tagged value 6-6, 6-36
{tgInsert} tagged value 6-6, 6-36
{tgUpdate} tagged value 6-6, 6-36
{type} tagged value 8-16, 9-20
<<check>> stereotype 6-9, 6-31
<<createView>> stereotype 6-10, 6-38
<<database>> stereotype 4-46, 5-10, 6-10, 6-29, 6-40
<<notNull>> constraint 5-13
<<notNull>> stereotype 4-25, 6-9, 6-31
<<null>> stereotype 6-9, 6-31
<<procedureClass>> stereotype 5-11, 6-9, 6-28
<<rule>> constraint 5-13, 5-16, 5-18
<<rule>> stereotype 6-9, 6-31
<<schema>> stereotype 4-46, 5-7, 6-10, 6-40
<<sqlPhysicalModel>> stereotype 6-10, 6-40
<<sqlPhysicalName>> stereotype 5-7
<<sqlView>> stereotype 5-11, 6-9, 6-28
<<storedProcedure>> stereotype 4-32, 6-10, 6-38
<<table>> stereotype 4-37, 4-38, 6-9, 6-28
<<trigger>> stereotype 6-10, 6-38
<<unique>> constraint 5-13
<<unique>> stereotype 6-9, 6-31
Abstract classes 4-5
Add a composite key 5-5
Add an index 5-5
Adding constraints 1-4
Adding triggers 1-4
Additional constraints 8-3
Adjusting tables 1-7
Analysis model 1-3
Annotating the model 5-3
Association annotations for Oracle
  {onDeleteCascade} tagged value 8-11
  Tagged values 8-11
Association annotations to Sybase
  {onDeleteCascade} tagged value 9-15
  Tagged values 9-14
Association unit
  Parameterization methods 7-17
  Signatures 7-17
  UML profile 7-17
Association unit method
  SetColumnPrefix 7-18
  SetPrimaryKeyName 7-18
  SetTableName 7-18
Associations
  "all" multiplicity 4-27
  Aggregations 4-25
  Cyclic associations 4-27
  Managing instances 4-27
Associations for Oracle
  Multiplicity constraints 8-10

Atomic attributes 4-16
Attribute annotations for Oracle
- {check} tagged value 8-13
- {null} tagged value 8-13
- Tagged values 8-13

Attribute annotations for Sybase
- {check} tagged value 9-18
- {null} tagged value 9-18
- {unique} tagged value 9-18
- Tagged values 9-17

Attribute unit method
- SetColumnName 7-20
- SetIsNullVarName 7-20

Attribute units
- Parameterization methods 7-19
- UML profile 7-19

Attribute visibility 4-17
Attributes
- Access restriction 4-17
- Atomic attributes 4-16
- Class attributes 4-17

Attributes for Oracle
- Mapping types 8-12

Attributes for Sybase
- Mapping types 9-16

Avoiding the update of the physical model 5-19
Basic types 4-16
Building a persistent schema 3-3
Calling on-line commands
- Commands which can be invoked 11-3
- Syntax 11-3

Class annotation for Sybase
- {primaryKey} tagged value 9-6

Class annotations for Oracle
- {check} tagged value 8-6
- {cluster} tagged value 8-6
- {indexSpace} tagged value 8-7
- {indexStorage} tagged value 8-7
- {initrans} tagged value 8-6
- {key} tagged value 8-6
- {maxtrans} tagged value 8-6
- {pctfree} tagged value 8-6
- {pctused} tagged value 8-6
- {storage} tagged value 8-6
- {tablespace} tagged value 8-6
- Tagged values 8-5

Class annotations for Sybase
- {check} tagged value 9-8
- {fillfactor} tagged value 9-8
- {key} tagged value 9-8
- {maxRowsPerPage} tagged value 9-9
- {partition} tagged value 9-9
- {schemaBD} tagged value 9-9
- {tablespace} tagged value 9-9
- Tagged values 9-7

Class association patterns 4-40
Class attributes 4-17
- Transformation 4-17

Class unit
- Parameterization methods 7-16
- Signatures 7-16

Class unit method
- SetPrimaryKeyName 7-16
- SetTableName 7-16

Classes
- Notes 7-21

Clusters 4-3, 8-3
Composite keys 5-5
Compositions

Application domain 4-36
Generated tables 4-35
Model and relational 4-34
Configuring the module 7-3
Configuring UML modeling projects 5-3
Constraints 1-3, 1-4
<<notNull>> constraint 5-13
<<rule>> constraint 5-13, 5-18
<<unique>> constraint 5-13
Creating a working environment for developing DDL scripts 2-4
Creating an SQL generation work product 3-9
Creating tables in a database 1-8
Creation scripts 1-7
Cyclic associations 4-27
Data Definition Language 1-3, 1-9, 3-11
Data mapping 1-8
Database diagram 1-8, 3-11, 4-14, 4-46, 5-3
DDL 1-9, 3-11
DDL scripts 1-3
Defining a SQL generation work product 3-10
Defining keys 1-6
Defining specific SQL generation characteristics 1-3
Defining tables 1-6
Designing a package 1-6
Destruction scripts 1-7
Developing applications
Clarifying tables 1-6
Creating tables 1-6
Designing a package 1-6
Generating the physical model 1-6
Dialog boxes 5-3
Distributed databases 4-46
{dbname} tagged value 8-14
Enterprise Edition 1-5
Environment variables 2-3
LANG 9-3
O_SQL_USER 2-3, 3-12, 8-3, 9-3
OBJING_PATH 2-3
PATH 9-3
SYBASE 9-3
Example of tables created in a database 5-27
Executing SQL 3-14
Executing the SQL
Procedure 5-26
Explorer 2-5
First steps
Building the database schema 3-14
Creating an SQL generation work product 3-9
Defining a SQL generation work product 3-10
Generating the physical model 3-6
Generating the SQL file 3-11
Importing the first steps package 3-4
Objective 3-3
System to be modeled 3-3
Visualizing the generated SQL 3-13
Free SQL sections
sqlAfter 7-21
sqlBefore 7-21
Generalization
Abstract class 4-15
Integrity 4-15
Mapping 4-5

Multiple generalization 4-15
Generalization and associations 4-42
Generalization and constraints 4-49
Generalization and the primary key 4-23
Generalization modes
One table per class 4-49
One table per concrete class 4-49
Generalization tree 4-5, 4-14
Generalizing associations with "One table per class" pattern 4-44
Generalizing associations with "One table per concrete class" pattern 4-43
Generalizing associations with "One table per generalization tree" pattern 4-45
Generalizing persistence 4-7
Example 4-7
Generalizing the generation mode 4-11
Generate physical model 5-4
Generate SQL files 5-4
Generated SQL
Visualizing 5-23
Generated SQL files 5-20
Generated SQL upper or lower case 9-21
Generating a class attribute in a specific table 4-20
Generating a physical model from a logical model 5-3
Generating creation scripts 1-7
Generating SQL code 1-7
Generating SQL corresponding to the physical model 5-3
Generating the physical model 1-6, 3-6
Different ways of generating the physical model 5-19
Generating triggers 4-28
Generation for Oracle
Environment variables 8-3
Generation for Sybase
Case sensitivity 9-4
End of the SQL command 9-3
Environment variables 9-3
Identifiers 9-4
Generation principles 1-8
Generation specific to Oracle
Primary key 8-4
Generation specific to Sybase
Parameterization 9-10
Referential constraints 9-10
Generation work product 1-9, 3-11, 3-15, 5-22, 5-27, 5-29
Grouping class attributes into tables 4-17
Grouping tables into clusters 1-4
Implementing a global table for class attributes 4-18
Import first steps 5-4
Indexes 4-57, 5-5
Integrity
Tagged values 4-53
Integrity checks 1-8
Integrity constraints
Consequences 4-48
not null attributes 4-53
Referential integrity 4-47
Uniqueness constraints 4-47
Interactive interface
Menus 5-4
Outlook 5-3

UML modeling project configuration 5-3
Interpreting SQL 5-3
J methods 7-19
SetForeignKeysName 9-11
SetForeignKeysNames 9-10
J methods for Oracle
Parameterization 8-8
SetForeign-KeysNames 8-9
SetForeignKeysNames 8-8
Keeping a generated SQL script 5-6
License 2-3
Logical model 1-3, 1-9
Man/Machine interface approach 1-8
Mapping 1-4, 1-9
{oneTable} tagged value 4-8
{oneTablePerClass} tagged value 4-8
{oneTablePerConcreteClass} tagged value 4-8
Choosing the right mapping technique 4-5
Example of {oneTable} 4-13
Example of {oneTablePerClass} 4-12
Example of {oneTablePerConcreteClass} 4-9
Examples 4-6
Mapping attributes in columns 4-16
Mapping class persistence 1-3
Mapping compositions 4-34
Mapping constraints 8-3, 9-3, 10-3
Mapping object model notions in relational database tables 4-3
Mapping rules 1-8, 4-3
Mapping rules for Oracle 4-16
Mapping techniques 4-5
All classes in a generalization tree mapped in a single table 4-5
Copying generalized attributes into all tables 4-5
The basic class has its own database table 4-5
Mapping types 8-3, 9-3, 10-3
Modeling 0..1-* relationships 4-24
Modeling a 0..1-* aggregation 4-25
Modeling a 0..1-* composition with an intermediary table 4-26
Modeling a 1-* association 4-26
Modeling a class attribute for the generation of a specific table 4-19
Modeling two class attributes 4-18
Module configuration 1-8
Module information 2-3
Module parameter sets 7-4
Multiplicity constraints 9-12
Naming conflicts 4-27
n-ary association pattern 4-38
Normal forms 4-16, 4-54
1NF 4-54
2NF 4-54
3NF 4-54
NF2 4-21
Note types on a class 6-7
Note types on a datatype 6-7
Note types on a package 6-8
Note types on an operation 6-7
Notes 2-5, 7-21
"sqlAfter" note 6-7, 6-28, 6-39, 7-21
"sqlBefore" note 6-7, 6-28, 6-39, 7-21
"sqlCode" note 5-16, 6-7, 6-32, 6-34, 6-37
Notes on a class 6-28

Notes on a datatype 6-32
Notes on a package 6-39
Notes on an operation 6-37
n-tuple 4-47
Object structure model 1-8
Objecteering/Administrating Objecteering Sites 1-5
Objecteering/Introduction 2-4
Objecteering/UML console 3-11
Objecteering/UML environment 1-5
Objecteering/UML Modeler 1-5, 2-5
Objecteering/UML Profile Builder 7-14, 7-21
On delete cascade 4-53
One table per class 4-44, 4-51
One table per concrete class 4-43, 4-49
One table per generalization tree 4-45
Optimization 8-3
Optimizing a class association 4-41
Oracle 1-3
Oracle mapping rules 4-16
Oracle model units
- Association 8-3
- Attribute 8-3
- Class 8-3

Oracle types
- Available types 8-16
- Parameterization 8-16

Outside keys 4-47
Package 4-46
- Designing a package 1-6

Package annotations for Sybase
- {genCascade} tagged value 9-5
- {schemaBD} tagged value 9-5
- Tagged values 9-5

Package unit
- Parameterization method 7-15

Packages
- Notes 7-21

Parameter sets
- Diagram generation on physical model 7-8
- External edition 7-13
- General 7-5
- Physical generation 7-6
- SQL generation 7-10
- SQL generation filters 7-11

Parameterization method
- Signature 7-15

Parameterizing Sybase generation
- Available types 9-20

Parameterizing the module 1-8
Pattern for taking into account associations of 1-1 type 4-37
Patterns for transforming 0..1-* associations 4-24
Persistence characteristics 1-3, 4-7
Persistent classes 1-3, 1-6, 4-15
Physical model 1-3, 1-9
Polymorphic reading 4-5
Predefined primitive classes 4-16
Primary key 1-9, 4-21, 4-35, 4-47, 4-54, 5-5, 8-11, 9-15
- Generation specific to Oracle 8-4
- Uniqueness 4-23

Primary key definition 4-21
Professional Edition 1-5
Propagation 3-12, 5-22
Properties editor 3-7, 3-12, 5-3, 5-23
- Adding notes 2-5
- Adding stereotypes 2-5
- Adding tagged values 2-5

Generating SQL 2-5
Items tab 3-12, 5-22, 5-24
Overview 2-5
Running SQL 2-5
SQL tab 1-6, 3-7, 5-19
Tabs 2-5
Visualizing generated SQL 2-5
Properties editor for SQL
On a class 5-11
On a datatype 5-16
On a logical model package 5-7
On a package inside a physical model 5-10
On a physical model attribute or a table column 5-18
On a physical model class or a class stereotyped <<table>> 5-17
On a physical model package 5-8
On an association 5-14
On an attribute 5-12
RDBMS 1-9
Relational database 1-3, 4-3, 4-54, 5-3
Relational database generation
General principles 4-3
Relational diagram 4-46
Relational table 1-9
Remote tables 8-14
Running scripts 1-7
Schema
Standardization 4-54
Scripts 2-4, 3-10
Secondary composite keys 4-57
Specific features of the RDBMS target 1-4
SQL
Execution 3-14
Generation procedure 3-11
Visualizing generated SQL 3-13
SQL database schema creation 1-9
SQL Designer commands 5-4
Add a composite key 5-5
Add an index 5-5
Composite keys 5-5
Execute alter table file 5-29
Execute SQL creation file 3-15, 5-27
Generate physical model 3-6, 5-4, 5-19
Generate SQL files 3-12, 5-4, 5-20
Import first steps 3-4, 5-4
Indexes 5-5
Primary key 5-5
Update from logical model 5-5
Visualize creation file 3-13, 5-24
SQL Designer commands on a class 5-5
SQL Designer commands on a package 5-4
SQL errors 9-19
SQL file 3-13
SQL generated for Sybase RDB
SQL errors 9-19
SQL generation
{onDeleteCascade} tagged value 9-13
SQL generation work product 3-9
Dialog box 3-10
SQL instructions 9-10
SQL script 4-16, 4-27
SQL scripts 5-6, 8-3
SQL Server 1-3
SQL Server model units
Association 10-3

Attribute 10-3
Class 10-3
Stereotypes 2-5
<<check>> stereotype 6-9, 6-31
<<createView>> stereotype 6-10, 6-38
<<database>> stereotype 4-46, 5-10, 6-10, 6-29, 6-40
<<notNull>> stereotype 4-25, 6-9, 6-31
<<null>> stereotype 6-9, 6-31
<<procedureClass>> stereotype 5-11, 6-9, 6-28
<<rule>> stereotype 6-9, 6-31
<<schema>> stereotype 4-46, 5-7, 6-10, 6-40
<<sqlPhysicalModel>> stereotype 6-10, 6-40
<<sqlPhysicalName>> stereotype 5-7
<<sqlView>> stereotype 5-11, 6-9, 6-28
<<storedProcedure>> stereotype 4-32, 6-10, 6-38
<<table>> stereotype 3-8, 4-37, 4-38, 6-9, 6-28
<<trigger>> stereotype 6-10, 6-38
<<unique>> stereotype 6-9, 6-31
Stereotypes on a class 6-9, 6-28
Stereotypes on a component 6-10, 6-29
Stereotypes on a constraint 6-9, 6-30
Stereotypes on a package 6-10, 6-39
Stereotypes on an operation 6-10
Stereotypes values on an operation 6-37
Stored procedure parameter types 4-33
Stored procedures 4-32
Structure 4-3
Sybase 1-3
Sybase generation
Parameterization 9-20
Sybase model units
Association 9-3
Attribute 9-3
Class 9-3
Tables 1-3, 1-4
Adjusting 1-7
Tagged values 1-3, 1-4, 1-6, 2-5, 4-3
{alterTable} tagged value 5-20, 5-28, 6-20
{check} tagged value 8-5, 8-11, 8-13, 9-7, 9-17
{classAttributesTableName} tagged value 4-19, 6-25
{cluster} tagged value 6-13, 6-23, 8-5, 8-11
{compositeKey} tagged value 6-20
{dbName} tagged value 6-29
{default} tagged value 8-13, 9-17
{external} tagged value 4-26, 5-15, 6-13
{fillfactor} tagged value 6-14, 6-23, 9-7, 9-14
{foreignKey} tagged value 4-25, 6-19
{genCascade} tagged value 9-5
{globalTable} tagged value 4-17, 6-19
{index} tagged value 4-3, 9-7
{indexKey} tagged value 4-57, 6-19

{indexName} tagged value 4-57, 6-27
{indexspace} tagged value 6-24, 8-5
{indexStorage} tagged value 6-25, 8-5
{initrans} tagged value 6-13, 6-23, 8-5, 8-11
{key} tagged value 8-5, 9-7
{keyName} tagged value 4-57, 6-26
{maxRowsPerPage} tagged value 6-14, 6-23, 9-7, 9-14
{maxtrans} tagged value 6-14, 6-24, 8-5, 8-11
{noDDL} tagged value 8-5
{not_null} tagged value 4-26
{null} tagged value 8-13, 9-17
{oid} tagged value 6-26
{onDeleteCascade} tagged value 4-48, 6-16, 8-11, 9-14
{oneTable} tagged value 4-8, 5-11, 6-25
{oneTablePerClass} tagged value 4-8, 5-11
{oneTablePerConcreteClass} tagged value 4-8, 5-11, 6-25
{partition} tagged value 6-13, 6-26, 9-7, 9-14
{pctfree} tagged value 6-13, 6-24, 8-5, 8-11
{pctused} tagged value 6-13, 6-24, 8-5, 8-11
{persistence} tagged value 4-7, 4-24, 4-40, 4-55, 5-11, 6-20, 6-27, 6-33
{primaryKey} tagged value 4-3, 4-21, 4-24, 4-37, 4-54, 5-15, 6-16, 6-19, 9-7
{sqlDefault} tagged value 4-20, 5-13, 6-19
{sqlDoNotUpdate} tagged value 4-37, 5-9, 5-19
{sqlName} tagged value 4-26, 5-7, 6-34
{sqlOptimized} tagged value 4-37, 4-40, 5-15, 6-16
{sqlType} tagged value 4-20, 4-33, 5-13, 6-20
{storage} tagged value 6-14, 6-24, 8-5, 8-11
{tablespace} tagged value 6-13, 6-26, 8-5, 8-11, 9-7, 9-14
{tgAfter} tagged value 6-36
{tgBefore} tagged value 6-36
{tgDelete} tagged value 6-36
{tgInsert} tagged value 6-36
{tgUpdate} tagged value 6-36
{unique} tagged value 8-13, 9-17
Tagged values on a class 6-5, 6-21
Tagged values on a component 6-6, 6-29
Tagged values on a generalization 6-6, 6-33
Tagged values on a model element 6-6, 6-34
Tagged values on an association 6-3, 6-11
Tagged values on an association end 6-3, 6-15
Tagged values on an attribute 6-4, 6-17
Tagged values on an operation 6-6, 6-35
Transforming 1-1 associations 4-37
Triggers 1-3, 1-4, 4-28, 8-10, 9-12
TD_name of the table 9-12
TI_name 8-10

TI_name of the table 9-12
Type mapping 4-16
Type project 1-9
UML profiles 4-42, 5-3, 9-10
Uniqueness constraint 4-21
Uniqueness constraints 4-47
Update from logical model 5-5
Updating a physical model 5-19
Different ways of updating a physical model 5-19
Updating a physical model from a modified logical model 5-3
Usage precautions 5-6
Using tagged values 4-3
Using the Objecteering/SQL Designer module 1-3
Visualizing produced scripts 1-7
Visualizing SQL 1-8, 5-3