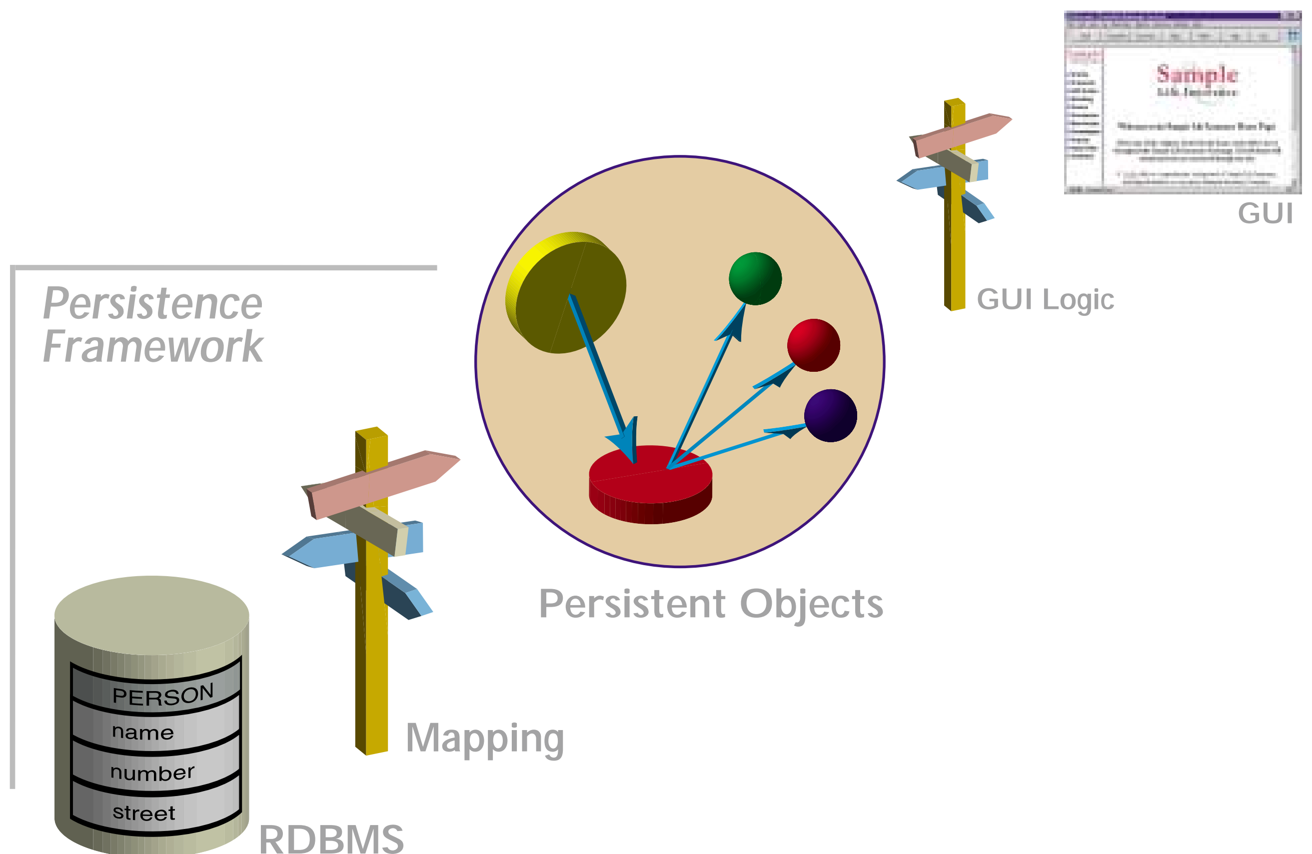
Persistence Framework

User’s Guide

For Frameworks Version 5.0



Copyright and trademarks

Copyright

Copyright 2000 Mynd. All rights reserved.

Mynd Frameworks V5.0.

Persistence Framework User's Guide, July 2000.

For more information about Mynd Frameworks, please contact:

Mynd SoftwareConsult GmbH
Taubenholzweg 1
D-51105 Köln (Cologne, Germany)

Tel. :	(+49) (0) 221 - 8029 - 0
FAX :	(+49) (0) 221 - 8029 - 999
Email:	info@mynd.de
Web:	www.mynd.com

Trademarks

ENVY is a registered trademark of the Object Technology International corporation.

Visual Age for Smalltalk and *OS/2* are registered trademarks of the International Business Machines Corp.

Windows and *Windows/NT* are registered trademarks of the Microsoft Corp.

Table of Contents

Copyright and trademarks *3*
Table of Contents *5*
Documentation overview *11*
Congratulations 11
Persistence Framework User's Guide (this manual) 11
Other manuals 12
Training from Mynd 13
Release Notes *15*
Version 5.0 15
Version 4.2 15
Version 4.0 15
Version 3.5 17
R3.4 V1.1 20
R3.4 V1.0 21
Installing the Persistence Framework *27*
System Requirements 27
Installation overview 27
Importing configuration maps 28
Loading application maps (with required maps) 28

1
Concepts and Architecture 29
1.1. Introduction 31
1.1.1. Requirements and Areas of Application 31
1.1.2. Basic Structure and Operation 32
1.1.3. Some examples 37
1.2. Static and dynamic POM components 38
1.2.1. Design changes to former releases: 38
1.2.2. Dynamic POM components 38
1.2.3. Static POM components 38
1.2.4. Sharing a POM-core between different POMs 39
1.3. Mapping Objects to Relational Data 41
1.3.1. Introduction 41
1.4. The Relational Data Model 42
1.4.1. Elements of the relational model 42
1.4.2. Normalization 42
1.4.3. SQL 43
1.4.4. Foreign key relationship properties 43
1.4.5. Some ramifications 44
1.5. The Object Model 46
1.5.1. Object model in general 46
1.5.2. The Smalltalk object model 46
1.6. From Relations to Objects (and vice versa) 47
1.6.1. The challenge 47
1.6.2. Mapping 47
1.6.3. Mapping Tool 47
1.6.4. Other tools 48
1.7. Starting an OO/RDBMS Project 49
1.7.1. Existing data model, new object model 49
1.7.2. Existing object model, new data model 49

1.7.3. Existing object model and data model ... 49
1.7.4. Activities ... 49
1.7.5. General patterns ... 50
1.7.6. Changing framework behavior ... 50

2
Tutorial ... 51
2.1. Introduction ... 53
2.2. The example: a customer base ... 54
2.2.1. Introduction ... 54
2.2.2. Requirements ... 54
2.2.3. Object and data model ... 54
2.3. Object mapping ... 57
2.3.1. Preparing the database ... 57
2.3.2. Setting up a POM ... 57
2.3.3. The main STOPF window ... 58
2.3.4. Instance variables ... 58
2.3.5. Derived classes ... 59
2.3.6. Internal table connection ... 59
2.3.7. Foreign key based relationships ... 59
2.3.8. Aggregation relationship ... 61
2.3.9. User-defined relationships ... 62
2.3.10. Optimistic concurrency ... 64
2.4. OQL ... 69

3
Mapping Guide and Reference ... 71
3.1. Introduction ... 73
3.2. Instance Variable to Column ... 74
3.3. Subclassing ... 75
3.3.1. Canonical subclass mapping ... 75
3.3.2. Denormalized subclass mapping ... 75
3.3.3. Unified subtable load ... 75
3.3.4. Subclass Table Connections ... 76
3.3.5. Type Discriminator ... 77
3.4. Classes Spanning Several Tables ... 78
3.5. Foreign key based relationships ... 79
3.5.1. Foreign key based 1-to-1 and 1-to-Many relationships ... 79
3.5.2. Many-to-Many relationships ... 80
3.6. Aggregated 1-to-1 relationship ... 81
3.7. User-defined relationships ... 82
3.7.1. What are they? ... 82
3.7.2. Dynamically defined user-defined relationships ... 82
3.7.3. Statically defined user-defined relationships ... 83
3.7.4. Defining relationship SQL clauses ... 83
3.7.5. Restrictions ... 84
3.7.6. Class MicFwRdbSQLClause ... 84
3.8. Special relationship properties ... 86
3.8.1. Cached relationships ... 86
3.8.2. Unidirectional relationships ... 87
3.8.3. Monitor target ... 87
3.9. Column Mapping Attributes ... 88
3.9.1. Value Strategy ... 88
3.9.2. Reload Strategy ... 88

3.9.3. Optimistic Strategy 89

4

Programming Reference 91
4.1. Introduction 93
4.2. Using a persistence manager 94
4.2.1. Get a global registered POM 94
4.2.2. Get a POM from an initializer class....94
4.3. Connecting to the database 95
4.3.1. Configuring the ODBC DLL....95
4.3.2. ODBC connection interface 95
4.3.3. ESQL connection interface 97
4.3.4. Oracle connection interface 97
4.4. The programming model 98
4.4.1. The transaction interface 98
4.4.2. The direct interface 101
4.4.3. The SQL Interface....102
4.5. Accessing objects 103
4.5.1. Introduction 103
4.5.2. Creating objects 103
4.5.3. Object retrieval....103
4.5.4. Relationships 106
4.5.5. Deleting objects 107
4.5.6. Changing objects 107
4.5.7. Application-specific extensions 107
4.6. Iterators 109
4.6.1. Description....109
4.6.2. Using iterators....109
4.6.3. Iteration exceptions....110
4.6.4. MicFwPsIterator -API 110
4.7. Miscellaneous topics 113
4.7.1. Sorting for relationships and extents....113
4.7.2. Object state transitions 114
4.7.3. Default database data types 114
4.7.4. Error handling 115
4.7.5. Empty strings stored in Oracle databases 116
4.8. Reentrancy 117
4.8.1. Registered objects 117
4.8.2. Iterators and Multi-Threading....118

5

Run-time POM 121
5.1. Introduction 123
5.1.1. What is an RT-POM....123
5.1.2. Why is an RT-POM necessary?....123
5.2. Requirements / restrictions / compatibility 124
5.2.1. Required maps....124
5.2.2. Requirements for application compatibility between Tool- and RT-POMs....124
5.2.3. Requirements for using an RT-POM in an application....124
5.3. Generate Run-time POM Options dialog 126
5.3.1. Options dialog....126
5.3.2. Tab Instanciation....127
5.3.3. Tab Code Generation 128
5.4. RT-POM initializer class 130

5.4.1. Overview 130
5.4.2. Creating 130
5.5. RT-POM and Runtime-Extended Descriptions 131
5.5.1. Benefits of RT-POM and RT-Extended descriptions 131
5.5.2. RT POM runnable with complete or RT extended descriptions 131
5.5.3. Method 1: Creating a POM out of a POM Storage (Initializer) Class or Export file 131
5.5.4. Method 2: Creating a POM out of a POM Storage (Initializer) Class or Export file 132
5.5.5. Notes 132

6
Object Queries 133
6.1. Introduction 135
6.1.1. Purpose of the query interface 135
6.1.2. An example query 135
6.2. The Query Block 136
6.2.1. Structure 136
6.2.2. Query variables 136
6.2.3. Path Expressions 136
6.2.4. Extent Expressions 137
6.2.5. Query Parameters 137
6.2.6. Literals 138
6.2.7. The Query Expression 138
6.3. Projection Queries 140
6.3.1. An Example 140
6.3.2. The Query Block 140
6.3.3. The Select Clause 140
6.3.4. Projection Operators 141
6.4. Alternative Query Interfaces 142
6.4.1. Array Interface 142
6.4.2. Stream Interface 142
6.5. SQL-Functions 144
6.5.1. Class hierarchy for supported SQL-functions: 144
6.5.2. How to build additional query functions 145
6.5.3. Usage of SQL-function with Stream Interface 145
6.5.4. Usage of SQL-function with projection queries (block-based OQL) 146
6.5.5. Usage of SQL-functions in projection queries (stream-based OQL) 146
6.6. Subqueries 148
6.6.1. Subqueries for Query Block Interface 148
6.6.2. Subqueries for Stream Interface 148
6.7. Error messages 150

7
Optimistic Concurrency 151
7.1. Introduction 153
7.1.1. What does optimistic concurrency mean ? 153
7.1.2. What are concurrency conflicts? 153
7.1.3. Dealing with conflicts 155
7.1.4. Configuring classes for optimistic concurrency 156
7.1.5. Handling update conflicts 158
7.1.6. Choosing the POM's commit handler 160
7.1.7. API methods 161

8
Embedded SQL 163

8.1. Introduction 165
8.1.1. What is Embedded SQL? 165
8.1.2. Dynamic and Static SQL 165
8.1.3. System Requirements 165
8.2. Architecture 166
8.3. Using Embedded SQL 168
8.3.1. Mapping for Embedded SQL 168
8.3.2. Application Development and Testing 168
8.3.3. Converting an existing POM to embedded SQL 168
8.3.4. Deployment 168
8.3.5. Connecting the POM 168
8.3.6. Mixing Dynamic and Static SQL 169
8.4. Tool Support 170
8.4.1. Precompilation and Code Generation Options 170
8.4.2. Parameter Select String Editor 171
8.5. Migrating from earlier Releases 173
8.5.1. Lazy Initializing of Statement Descriptors 173
8.5.2. Multiple Module Support 173

9

Cross Development Support 175
9.1. Introduction 177
9.2. Setting up for Cross Development 178
9.3. Using the Persistency Frameworks in an XD Enviroment 179
9.3.1. Application Development and Testing 179
9.3.2. Store a POM in a target image 179
9.3.3. Precompile a POM in a target image 179

10

Tools 181
10.1. Introduction 183
10.2. The STOPF tool 184
10.2.1. Overview 184
10.2.2. Browsing a POM 184
10.2.3. Creating a POM 185
10.2.4. Deleting a POM 185
10.2.5. Inspect a POM 186
10.2.6. Storing and exporting a POM 186
10.2.7. POM Configuration Editor 187
10.2.8. POM ODBC options 188
10.2.9. Statement Registration Tool 189
10.2.10. Validation, compilation and method generation 193
10.2.11. Editing mappings and relationships 196
10.2.12. The Class menu 196
10.2.13. The Variable menu 197
10.2.14. User-defined relationship 200
10.2.15. Setting column mapping attributes 203
10.2.16. Editing a database schema 204
10.2.17. Exporting a database schema 206
10.2.18. Mapped class generation 207
10.3. The POM Generator Tool 209
10.3.1. POM generator 209
10.4. Interactive SQL 213
10.4.1. Overview 213

10.4.2. Usage 213

Appendix A

API 215

Appendix B

Glossary 241

Bibliography* *249

List Of Tables* *251

List Of Figures* *253

Index* *255

Documentation overview

Congratulations

Linking object-oriented programming and database technology is now one of the greatest challenges in the development of business-oriented DP solutions. The development of applications using object-oriented methods is becoming increasingly important. However, database technology (typically relational) is usually employed for storing corporate data. The break between the paradigms in application development and data management causes considerable problems, making data-processing difficult if not impossible.

The Persistence Framework from Mynd is just one of the powerful tools in the Mynd Frameworks. The Persistence Framework is a powerful Smalltalk development environment that provides the required functionality for implementing your database applications.

Your purchase of the Mynd Frameworks reflects your uncompromising insistence on the most advanced object–oriented technology. The functionality and performance of the Persistence Framework is a result of Mynd's extensive project experience in the banking and insurance industries.

The Persistence Framework documentation from Mynd is designed to help you exploit the full potential of the Persistence Framework tool in the shortest time possible.

We wish you success in implementing your projects with the Persistence Framework.

The Mynd Frameworks team

Persistence Framework User's Guide (this manual)

Intended audience

This manual is aimed at software developers who want to produce database-oriented applications using the micado Frameworks, especially the micado Object Persistence Framework. It explains the basic concept, tools and programming facilities of the Framework, and shows how it can help increase productivity and contribute to the development of applications for long-term storage, processing and re-utilization of business objects. This manual is not a text book, but rather is geared to the practical needs of the developer.

One requirement of object-oriented applications development is that the software can be reused and easily adapted. And this applies equally to the Framework which is in a continuous state of further development and adaptation. This manual therefore "only" represents a milestone in the development of the Framework. New developments and changes in the future will be reflected accordingly in the documentation (e.g. loose-leaf appendices with descriptions of the new or modified functionality).

We have assumed that the reader is familiar with the Smalltalk programming language and the concepts and programming techniques on which it is based. We have also assumed that he has a basic knowledge of databases, especially relational databases. Although the Object Persistence Framework essentially conceals the database interface from the application developer, knowledge of database technology is vital for achieving an optimum result when bringing both worlds together, especially to the creator of the object and data model. A short refresher course on some of the most important concepts in relational databases therefore appears in 'The Relational Data Model' (page 42). To gain a deeper understanding of this material, we recommend that you study relevant literature and participate in appropriate training courses.

Since the concept and technology of the Persistence Framework are based on the Object Behavior Framework, we have also assumed that the reader is familiar with these (see the Object Behavior User's Guide).

Sections

This manual contains the following sections:

- **'Copyright and trademarks' (page 3)**.
- **'Table of Contents' (page 5)**.
- **'Documentation overview' (page 11)** (this section).
- **'Release Notes' (page 15)**.
- **'Installing the Persistence Framework' (page 27)**.
- **'Concepts and Architecture' (page 29)**. Explains the major concepts for the Persistence Framework.
- **'Tutorial' (page 51)**. Provides a set of programming examples that demonstrate the major programming concepts and techniques of the Persistence Framework.

- 'Mapping Guide and Reference' (page 71). Gives a detailed insight into the special features of mapping Smalltalk classes and objects onto relational tables as offered by ther Persistence Framework. The different mapping constructs and the data modeling and programming implications resulting from their use are explained. This chapter is especially relevant for personnel who are involved in the job of "mapping", i.e. defining the mapping rules between the models. It can also serve as a reference for the material in the chapter dealing with development tools.
- 'Programming Reference' (page 91). Provides a detailed account of the facilities and requirements for programming with the Framework. This is achieved by describing the available programming interfaces (APIs) and the use of "mapped" objects. This part of the manual should serve as a reference for specific problems arising during programming. However, it would also be advisable to obtain an overview of the subjects which are dealt with in this chapter before starting a project.
- 'Run-time POM' (page 121). Contains a brief explanation of what a runtime-POM is, what its benefits are and how to use it.
- 'Object Queries' (page 133). Describes the Persistence frameworks Object-Query-Language with the different alternative syntaxes and the concept behind it.
- 'Optimistic Concurrency' (page 151). Explains the concepts and configurations that the framework offers to support a multi-user environment with optimistic concurrency.
- 'Embedded SQL' (page 163). Gives a description about the static (embedded) SQL concept in general followed by the embedded SQL support that the framework provides. A description of the framework tools to work with embedded SQL is also part of this chapter.
- 'Cross Development Support' (page 175). Explains the tools that help you to use the framework in a cross developement environment with passive images, e.g. when using MVS Smalltalk.
- 'Tools' (page 181). Explains the development tools belonging to the Framework and their integration into the Smalltalk development environment. The "STOPF" mapping tool is the most important component of the development environment, since it effectively supports the mapping of the Smalltalk object model onto the relational data model using a graphical user interface and comprehensive plausibility checking. Together with the previous one, this chapter should be read by personnel who are involved in the creation of mapping rules.
- **'API' (page 215)**. Describes the API methods for the Persistence Framework.
- **'Glossary' (page 241)**. Includes any special terms and abbreviations used throughout this manual.
- **'Bibliography' (page 249)**.
- **'List Of Tables' (page 251)**.
- **'List Of Figures' (page 253)**.
- **'Index' (page 255)**.

Conventions used in this manual

The following conventions are used in this manual.

- A term being used for the first time is bold and italic. For example: ***ODMG93 standard***.
- Dialog names and menu entries are displayed in bold, with a forward slash between nested entries. For example: **System Transcript / Tools / Manage Applications**.
- Smalltalk code appears in fixed-width Courier format. For example:

```
^self
```

- SQL statements appear as capitalized text in fixed-width Courier format. For example:

```
CREATE TABLE CUSTOMER
```

- In relational schemata, relationships are shown in their "natural" tabular format with column headings. Key column names are underlined, and foreign key column names appear in italics. A "hash" sign (#) after a column name is the abbreviation for Number. For example, Kd# means "customer number"

Reader comments

Any comments you have concerning this manual can be sent to:

info@mynd.de

Other manuals

From Mynd (and referenced throughout this manual):

- Frameworks Getting Started Manual.
- Application Framework User's Guide.
- Object Behavior Framework User's Guide.

From IBM:

- IBM Visual Age for Smalltalk manuals.

Training from Mynd

This manual is provides complete information for the Persistence Framework. However, it is also recommended to consider enrolling in a training course from Mynd that is customized to your specific needs. Mynd provides a wide-range of courses covering Smalltalk programming, object-oriented analysis, design, methodolgy, and project management.

For more information about training courses, please contact:

Mynd SoftwareConsult GmbH
Taubenholzweg 1
D-51105 Köln (Cologne, Germany)

Tel. :	(+49) (0) 221 - 8029 - 0
FAX :	(+49) (0) 221 - 8029 - 999
Email:	info@mynd.de
Web:	www.mynd.de

Release Notes

Version 5.0

1. VisualAge 5.01 compatibility

Frameworks (including Persistence Framework) now run under VisualAge 5.01.

Version 4.2

1. Relationship invalidate error fixed

A relationship can now be invalidated without invalidating other relationships with the same source and target class that were created in the same context before the invalidate message was sent.

In the previous version, invalidating such a relationship would cause the invalidate message to be sent for the other relationships. The result would be the deletion of the other relationships.

Version 4.0

1. New ClassMapping Option: Cached

Every mapped class in a POM can be set to "cached" (STOPF: Class->Cached).

If a reference is made to a "cached" class as a target of a :1 or 1:N relationship, then the object returned is not from the database, but rather from the Collection that the cached target class returns in response to the message #cachedObjects (class method). This is similar to cached relationships (see **'Cached relationships' (page 86)**).

Example

Assume the following 2 classes exist:

- Class City
- Class Country

There is a 1:N relationship between the classes:

- Country>>cities <--->> City>>myCountry

Assume the class "Country" is set to "cached". The following code

```
| targetCountry |
targetCountry := aCity myCountry
```

would cause the message "cachedObjects" to be sent to the class Country. This requires that:

- The cached classes understand the message "cachedObjects"
- All instances of this class are returned as a Collection

The Framework resolves the relationship by returning the required object from the Collection.

Assume that in the above example class "City" is also marked as "cached". Then the following code:

```
| cityList |
cityList := aCountry cities asArray
```

would cause the message "cachedObjects" to be sent to the class City. In response the Frameworks would return all required objects from the resulting Collection of City objects.

Notes

Option "cached" has no effect when loading via queries (OQL, etc.) or PersistenceIdentifier (with primary keys).

Option "cached" is currently not supported for N:M relationships (loading is from the database).

To optimize the search for the target object of a relationship, overwrite the following methods via a subclass of the PersistenceManager class that is used:

- MicFwAbstractPersistenceManagerRdb>>cachedObjectsFor:with:from:
- MicFwAbstractPersistenceManagerRdb>>cachedObjectsListFor:with:from:

Normally the Framework seeks the required objects sequentially in the cachedObjects Collection. By overwriting these methods it is possible to:

- Find the object(s) faster using hashing

  or
- Use different cachedObjects Collection's for different POMs.

2. New menu item for creating a POM

A new PersistenceManager is created by:

1.Selecting **Transcript / micFrameworks / Browse Persistence Manager**.

2.Selecting **<New Persistence Manager>** from the list.

The menu item **Transcript / micFrameworks / New Persistence Manager** no longer exists.

3. Optimistic concurrency enhancements

The following enhancements were made for optimistic concurrency.

3.1. Parent Deleted Conflict Exception #MicFwPsParentDeletedError

If a parent object (for example: Employee) of a :1 relationship is deleted and then the accessor (for example: ownerEmployee) of a child object (for example: Customer) of the parent is executed, a **#MicFwPsParentDeletedError** exception is generated (previously #nil was returned without generating an exception).

Example

```
testCaseRelationshipParentDeleted
   "test the MicFwPsParentDeletedError exception, that is thrown when
   the parent of a relationship was deleted before resolving the relationship
   of the dependent object"

   | context  emp1 emp2 cust result1 result2 |

   result1 := false.
   result2 := false.

   [  self connectToDataSource.

      "delete old testData"
      context := self pom newPersistenceContext.
      context beginTransaction.
      self pom
         deleteInstancesOf: MicFwTutorialCustomer where: [:c | c id =3];
         deleteInstancesOf: MicFwTutorialEmployee where: [:e | (e id = 1) or: (e 
id = 2)].
      context commitTransaction.
      self pom resetCache.

      "create test data"
      context := self pom newPersistenceContext.
      context beginTransaction.
      emp1  := MicFwTutorialEmployee newPersistent  id: 1; yourself.
      emp2  := MicFwTutorialEmployee newPersistent  id: 2; yourself.
      cust  := MicFwTutorialCustomer newPersistent  id: 3; yourself.
      cust ownerEmployee: emp2.
      context commitTransaction.

      "load customer"
      cust := (self pom persistenceIdentifierFor: MicFwTutorialCustomer) set-
ValueFor:  #id to: 3; load.

      "simulate, that an other user change the employee of the customer and 
delete
      the previous referenced employee"
      self pom
         executeSQL: 'update customer set employeeId = 1 where id = 3';
         executeSQL: 'delete from employee where id = 2';
         executeSQL: 'delete from person where id = 2';
         commitDatabaseTransaction.

      result1 := (cust persistenceProperties fkValues at: #ownerEmployee) = 
#(2).
```


MYND

```
     "the following should produce an exception, because the fk-value in the
persistenceProperties
     of the customer is = 2, but no employee with the id = 2 can be loaded "

     [cust ownerEmployee id]
        on: MicFwPsParentDeletedError do: [ :exception | result2 := true].
    ]
   ensure: [
     "delete test data"
     context := self pom newPersistenceContext.
     context beginTransaction.
     self pom
       deleteInstancesOf: MicFwTutorialCustomer where: [:c| c id =3];
       deleteInstancesOf: MicFwTutorialEmployee where: [:e| (e id = 1) or: (e
id =2)].
     context commitTransaction.
     self pom resetCache.

     self abortTransactionAndDisconnectFromDataSource: context].

   ^result1 & result2
```

Version 3.5

1. Changed APIs

#createExtendedDescription methods must be migrated before PersistentObject-classes stored with former releases.

Removed methods

`MicFwPersistentObject class>>#removePersistence: (micInternal, D)`

Changed behavior

`MicFwPersistentObject>>#allPersistentRelationships`
`MicFwPersistentObject>>#allPersistentRelationshipsDo:`

Relationships

Former releases: All relationships that have the 'persistent' flag set in the object net browser. The methods worked on the static flag defined on the class level.

Current release: All relationships, that are currently persistent in the receiver, e.g.=> the relationship must be mapped in the receiver's current POM.

2. New POM Validations

WARNING: The foreign key <fk-name> is mapped is multiple ways at <class name>-><relationship-aspect> <class name>-> <relationship aspect> so that relationships might not be loaded correctly!

3. Inspect POM

The menu Transcript->micFrameworks->Persistence Tools->Inspect POM has been renamed to Transcript->micFrameworks->Persistence Tools->Inspect Tool-POM. The list now also includes Tool-POM-Initializer classes.

The menu Transcript->micFrameworks->Persistence Tools->Inspect RT-POM is new. It allows you to inspect RT-POM instances stored globally in the image or stored in a RT-POM Initializer class.

4. Native Database Interfaces

Introduction

The Mynd Native Database Interface used a DLL to access the Database. It promised better performance and a smaller Image compared to other POMs. The behavior of a MicFwPersistenceManagerRdbNative POM is almost the same as that of its Non-Native counterpart.

System requirements

Native Database Interface is currently supported for DB/2 Version 5.0 or later and Oracle Version 8.0 or later. The DB/2 Interface is currently available for Windows and OS/2. The Oracle Interface is currenty only available for Windows.

Installation

Copy the appropiated DLL into a dictionary where the Smalltalk image resides or in a dictionary of the search path.

```
DB/2:     micdb2.dll
Oracle:   micora.dll
```

For a Runtime environment you must load one of the following Configuration Map:

```
DB/2:     micPersistenceFrameworkNativeDB2 Runtime
Oracle:   micPersistenceFrameworkNativeOracle Runtime
```

In a Deveolpment environment you can additional load one of the following Configuration Map:

```
DB/2:     micPersistenceFrameworkNativeDB2 Development
Oracle:   micPersistenceFrameworkNativeOracle Development
```

You can validate your Installation with the following Code:

```
DB/2:     MicNativeDb2DatabaseConnection validatePlatformFunctions
Oracle:   MicNativeOracleDatabaseConnection validatePlatformFunctions
```

Migration

You can migrate an existing POM to the appropiated Native-POM using the Native-POM method #createAndStoreFrom:. For example:

```
MicFwPersistenceManagerNativeOracle createAndStoreFrom:
   (MicFwPersistenceManagerOracle named: 'MicFwORACLETestPOM')
```

Error trace / Error report

The C-DLL can generate a trace which can be enabled or disabled at any time.

Turn on trace

The parameter for method #callStartTrace: is the name of the trace file.

```
DB2:    MicNativeDb2DatabaseConnection new callTraceStart: 'mic'
Oracle: MicNativeOraveDatabaseConnection new callTraceStart: 'mic'
```

Turn off trace

```
DB2:    MicNativeDb2DatabaseConnection new callTraceStop
Oracle: MicNativeOraveDatabaseConnection new callTraceStop
```

DB/2

A CLI-Trace is always helpful when search for the source of an error. A CLI-Trace can be enabled / disabled by an entry in file db2.cli (in DB2 directory \SQLLIB):

```
[Common]
;  Path must exist
TRACEPATHNAME=D:\gko
TRACEFLUSH=1
; 1 = Trace on 0 = Trace off
TRACE=1
```

Trace can be disabled at any time with TRACE=0.

If the Trace is enabled: A file will be created for each connection. Every CLI call will be registered in this file. The Trace slows execution speed significantly!

When reporting errors: Please attach the relevant files.

5. SQL-Editor enhancements

In some cases changing framework-generated SQL statements can result in better performance (e.g. changing the order of the terms in the where clause).

An editor for the statements can be opened from the Statement Registration Tool. After selecting one statement in the list use 'Edit Generated SQL' from the context- or the 'statements'-menu. If the statement is a projection query or an OQL/SQL extend, the statement's type (load, count or delete (delete only at OQL extend)) has to be chosen. If the statement holds more than one SQL statement (if the class is mapped to more than one table) a list for selecting the SQL statement appears.

There are no restrictions when editing the SQL statement's string. The syntax will not be checked. It is important to change the parameters analogous to the changes in the SQL string. The parameters can be duplicated or deleted and the order can be changed. The columns in the list of parameters have the following meanings:

- Index: parameter's index in the registered statement
- Table: mapped table's name
- Column: column's name in the mapped table
- Value/Name: display either the constant value, that will be used at execution of the SQL, or the name of

the parameter. The abbreviations in front of the parameter names show the type of the parameter (P=Value Parameter, AP=Aggregate Parameter, OP=Object Parameter). For base statements this column is mostly empty (except of subtype columns).

- Index in Default: parameter's index in the default statement.

Use the 'Default' button to set the SQL statement and all parameters to the framework generated default values. The 'Register' button registers the statement and closes the editor. If the statement or its parameter have been changed, this will be indicated in the Statement Registration Tool by a 'U' (= user defined) in the 'Pregenerated' column. The statement is indicated as user defined if at least one SQL statement is not the default.

6. Runtime-POM generator enhancements

The Runtime-POM (RT-POM) is a new feature that was first included in a redesigned form in release R3.4 V1.1. The RT-POM instance should be stored in the image, inspected or you could dump the instance with the Object Dumper.

Options dialog

The RT-POM generator browser offers you more possibilities in R3.5 to distribute and store the RT-POM. You can decide whether you want to generate a RT-POM instance directly (Instanciation) or if you want a RT-POM intializer class (Code Generation) to be generated.

Tab Instanciation

The "Instanciation" page now contains a "Dump to File" option, which writes the RT-POM instance to a dump-file using Object Dumper.

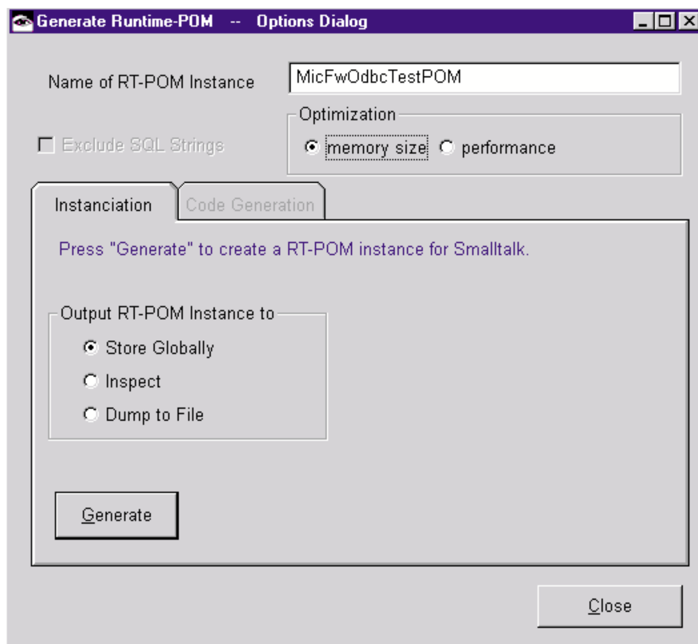


Figure 1. RT-POM dialog Options tab Instanciation

Tab Code Generation

Release R3.5 allows to store a RT-POM in an initializer class.

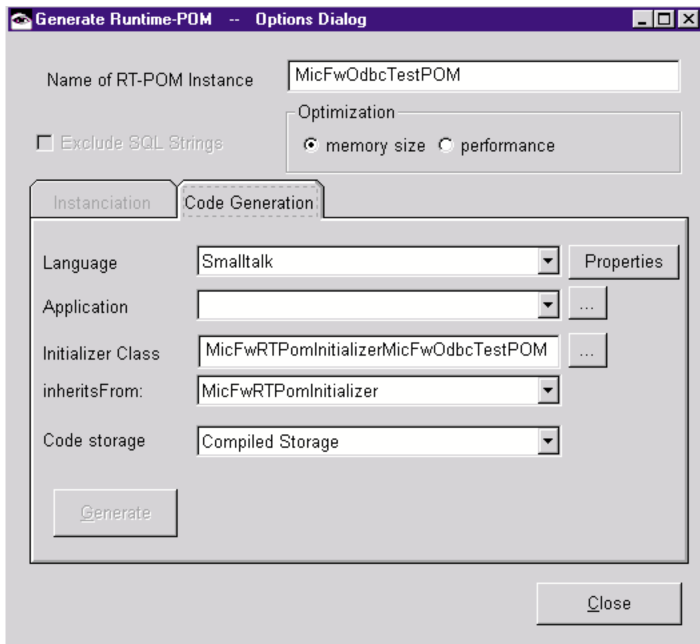


Figure 2. RT-POM dialog Options tab Code Generation

RT-POM initializer class

RT-POM initializer class makes it more flexible and easy to use the RT-POM in your packaged application.
Without a RT-POM initializer class you had two possibilities:

- Transfer the RT-POM instance to packaged image with the VA-packager
- Application could load the RT-POM from a dump file

With a RT-POM Initializer you can:

- Create the RT-POM Instance quickly within your application.
- Distribute a RT-POM save and quickly between other developers in the team.

7. POM Generator enhancements

The following enhancements have been made to the POM Generator:

- Naming conventions for tables and columns can be defined (menu **Options / Edit Name Patterns** in POM Generator).
- Class->DB type mappings can be customized (button **Customize Mappings...** in POM Generator).
- Bugs that occur in unusual configurations have been fixed.
- The order in which types are assigned for predefined types has been changed: The shortest types are assigned first (not the longest types). For example: For String without a defined length: CHAR is assigned before VARCHAR).

8. STOPF tool enhancements

The following enhancements have been made to the STOPF tool:

- Additional options for schema export (generate DROP TABLE statements, INDICES for Primary keys, etc.)
- Naming conventions for primary key indices can be defined.
- New menus have been added, including:
  - Find Table
  - Convert To
- Conversion of a tool-POM from 1 POM class to another is possible (for example: converting from MicF-wPersistenceManagerODBC to MicFwPersistenceManagerNativeDB2).

9. OQL subquery fixes

The query result of the outer query can now be accessed in subqueries.

R3.4 V1.1

1. Event #sqlConnect:options: removed

MicFwPersistenceManagerOdbc does not support the event #sqlConnect:options: anymore.

2. Redesign of POM-architecture

In R3.4 V1.1 the POM-architecture and mapping classes have been redesigned to fulfill the following requirements:

- Reusage of POM-components in different POMs (support for client specific state in different POMs)
- Reentrancy

See the new chapters about the Runtime-POM and reentrancy issues for details ‘Static and dynamic POM components’ (page 38).

3. Registered objects

The registry in the POM-core can hold all kinds of objects that implement the Registerable interface. These are:

- OQL-Queries (MicFwPsExtentIdentifier)
- SQL-Queries (MicFwPsExtentIdentifierSQL)
- OQL-Projections (MicFwPsProjectionDescriptor)
- Simple SQL-Statements (MicFwRdbDirectStatement)
- OrderClauses (MicFwRdbOrderClause)

In former releases the use of registered objects was not thread-safe, because the POM returned the registered instance and not a copy. In the current release, the POM returns a copy of the registered object, if neccessary where the dynamic components are copied. So it is possible to use the same registered object for multi-threaded execution, when each thread works with an own copy.

See the new chapter about reentrancy for details: ‘Reentrancy’ (page 117).

4. Runtime-POM technology

The code-generation based runtime-POM (called OptimizedPomSupport) is no longer available for the persistence frameworks.

The reason to drop the OptimizedPomSupport was the large amount of memory required for the many generated methods the OptimizedPOM-Instance consisted of.

Instead of the code-generation based approach, you can create a runtime-POM with another tool and another concept behind it.

The new RT-POM solution is implemented in the following new appliations, that are part of the frameworks in R3.4 V1.1 for the first time:

- MicFwRdbRTMappingInfos
  Neccessary runtime classes for RT-POMs (includes RT-mappings)
- MicFwRdbRTMappingInfosInit
  Initialization methods for RT-POMs (mapping initialization methods). This application is not neccessary during runtime.
- MicFwRdbRTMappingInfosD
  Developlement classes and methods. (includes the new RT-POM generator)
- MicFwRdbRTMappingInfosTools
  GUI Tools (includes the new RT-POM generator browser and GUI)

For more details and a description of how to use the tools see chapter ‘Run-time POM’ (page 121).

R3.4 V1.0

1. Removed OBSOLETE POM-Classes

The obsolete POM-classes have been removed (from MicFwPersistenceOdbcObsolete application).

If your application still used one of these classes (by subclassing or having a POM instance) you have to switch to a current POM configuration as described below:

STEP 1: Preparation

Make sure that you have created a "POM storage class" (STOPF→Manager→Store in class) or a "POM Export file" (STOPF→Manager→Export to file) with the old release before you migrate to the new release.

STEP 2: Switch POM class

If you have subclassed the obsolete classes:

Before you instanciate your POM, change the superclass of your subclass to the class mentioned in column "Use this class instead:" in the table below. Then instanciate the POM from the "POM storage class"or the "POM Export file".

If your POM is an instance of an obsolete class:

If you have stored the POM in a "POM storage class":

Browse your POM storage class (it is a subclass of MicFwPersistenceManagerInitializer) and modify the class method `#pomClass`, so that it answers the class mentioned in column "Use this class instead:" in the table below instead of the obsolete POM class.

Then instanciate the POM from this initializer class.

If you have stored the POM in a "POM Export file":

Open the POM Export file and modify the first Smalltalk statement

```
Smalltalk
at: #MicFwManagerImportGlobal
put: (MicFwPersistenceManagerOdbc new
storeGloballyAs: 'MyPomName';
yourself)!
```

to refer to the POM class mentioned in column "Use this class instead:" in the table below.

STEP 3: Adapt POM configuration

Browse your POM (Transcript→micFrameworks→Browse Persistence Manager) and open the POM configuration editor (STOPF→Manager→ Configuration).

If you used the default configuration:

Choose "Configurations→Select..." in the configuration editor and apply the configuration mentioned in column "Use this configuration instead:"

If you have customized the configuration of your POM:

Refer to column "Specifics of obsolete class:" to adapt the POM configuration to your needs.

Save your changes before you close the STOPF tool in a "POM Export file" or "POM class storage".

Obsolete class (removed)	Specifics of obsolete class:	Use this class instead:	Use this configuration instead:
MicFwPersistenceOdbcDb2	Database Description: MicFwDb2Odbc	MicFwPersistence-ManagerOdbc	"Odbc (ODBTalk) POM for DB2 Version 1.2 and 2.1" or "Odbc (ODBTalk) POM for DB2 Version 5 (IBM DB2 Universal Database)"
MicFwPsManagerOdbcDB22	Database Description: MicFwDb2Odbc		
Statement Executor: MicFwRdbStmtExecOdbcDB212	MicFwPersistence-ManagerOdbc	"Odbc (ODBTalk) POM for DB2 Version 1.2 and 2.1"	
MicFwPsManagerOd-bcOracle	Database Description: MicFwOracleOdbc	MicFwPersistence-ManagerOdbc	"Odbc (ODBTalk) POM for Oracle"
MicFwPersistenceManagerOdbcWatcom	Statement Executor: MicFwRdbStmtExec-OdbcWatcom	MicFwPersistence-ManagerOdbc	"Odbc (ODBTalk) POM for Sybase SQL Anywhere"

Table 1: Obsolete classes.

2. POM-Generator fixes

Fixes and enhancements in the POM-Generator-Tool:

Support for primitive to-N relationships

The POM-Generator supports primitive to-N relationships by generating an associative table (as to N-to-M relationships)

Support for redefined variables

The POM-Generator now considers redefined variable typings. This means, that a new column and mapping will be generated for aspects, that redefine the typing from their superclass.

CAUTION: To avoid backward compatibility problems, you can print a list of redefined variables with the new menu item: Transcript→micFrameworks→Object Behavior Tools→Find Redefined Variable Typings...

It happens, that some variables become redefined during development without necessity. This can cause some unpleasant surprises with the POM Generator.

You can remove a redefinition with the Object Net Browser's menu item:

Variable->Remove Description

Suppress table for classes with primitive relationships

The POM-Generator is able to suppress a class, that has primitive relationships.

Support for aggregated 1-to-1 relationship mappings

The POM-Generator can aggregate a class that has a 1-to-1 relationship to its enclosing class.

The POM-Generator can aggregate a class that has aggregated subclasses (no subtyping).

3. Unified subtable load

An alternative load algorithm has been implemented for subtype classes, that can be defined for each class mapping.

You have the following options:

- Tablewise subtable load (the default, as usual)
- Load with a UNION Statement (=Unified subtable load)

For more details refer to chapter 'Unified subtable load' (page 75).

4. ESQL Performance enhancements

This release offers enhanced performance for embedded SQL (ESQL) through the following means:

- Descriptor lazy initialization (descriptor information is created and cached when the method is accessed the first time)
- Methods are accessed from a preallocated Array through indexes instead from an IdentityDictionary

This yields the following advantages:

- For large POMs initialization tme is reduced.
- since often only a small subset of the generated methods is actually used, overhead for unused methods is avoided
- Access to the methods through an indexed array is much faster
- The array is preallocated with the (maximum) required size, thus avoiding expand/copy operations

In order to achieve these benefits, do the following:

- Delete the method `#initializeModulesFor:` from any previously generated MicFwEsqlAccessSet subclasses.
- Delete or unload any previously generated MicFwESQLModule subclasses
- Load the current Frameworks release
- Precompile all POMs as usual

The POM archival code will only need to be saved is the name of the AccessSet class was changed.

5. Application MicFwStmtHdlrInnerJoin removed

The Application MicFwStmtHdlrInnerJoin, that was neccessary for all common FatClient or Server RdbPoms and was probably used as a prerequiste by your application no longer exists!

Migration:

Remove MicFwStmtHdlrInnerJoin from the prerequisites of your applications and set a prerequisite to MicFwPersistenceRdb instead.

6. Classes MicFwDB212 and MicFwOdbcDb212 removed

Database Description classes MicFwDB212 and MicFwOdbcDb212 no longer exist. These classes existed only in interim releases R3.3 V2.6 and R3.3 V2.7.

Migration:

If you have used one of these classes to configurate EXISTS or IN clauses for SQL statements generated from OQL-expressions, you can use database description classes MicFwDB2 or MicFwOdbcDb2 in your POM and configure the POM as described in chapter 'POM Configuration Editor' (page 187).

7. New POM Configuration Option: "Generate EXISTS"

The POM Configuration Editor shows a new option named "Generate EXISTS".

This option decides whether SQL-Statements that the Persistence Framework generate from OQL-expressions (Object-Query-Language) use EXISTS-clauses or IN-Clause in the WHERE-part of the SQL.

Older versions of the OQL used EXISTS-clauses only, that lead to performance problems with some DBMS's. The current version uses IN-clauses by default.

8. No INSERT-statements for classes with unmapped NOT NULL columns

The persistence framework will no show INSERT-statements for pregeneration in the Statement Registration Editor, for classes that do not map all of the NOT NULL columns in the tables they are mapped to. E.g. this means that abstract superclasses with a subtype column declared with NOT NULL will not have an insert-statement pregenerated.

In former releases this was not checked by the Frameworks Development tools with produces runtime-errors during ESQL-precompile time, because the DBMS throwed an integrity violation for such an insert-statement.

("Assignment of a NULL value to a NOT NULL column is not allowed. SQLSTATE=23502")

For details see methods:

- MicFwRdbClassMapping>>isValidForInsert (in application MicFwPersistenceRdbD)
- MicFwStatementCategory>>forInsert: (in subapplication MicFwRdbStatementCategorization)

9. Behavior changed: Loaded state of relationships

In releases up to R3.3 V2.4 the behavior was the following:

A relationship became loaded (= resolved from DB so that it won't touch the DB again) after

- it has been send the method #getTarget
- it is a NRelationship and has been send the method #delete
- after commitTransaction when it is a 1Relationship mapped on the foreign key side (FKDependentMapping)
- after commitTransaction when it is an aggregated 1Relationship (AggregateMapping)
- after the object has been loaded from the DB when it is an aggregated 1Relationship

To spare some DB accesses and to have an equal behavior for standalone and client POMs we changed it in the following way.

In releases newer than R3.3 V2.4 a relationship becomes loaded when:

- 1. - 5. the same as above
- after commitTransaction when the owner object was newly inserted into the DB (This means that all relationships for newly inserted objects are loaded afterwards.)
- after commitTransaction when the relationship has been changed/versioned and it is a 1Relationship (for both Dependent and Parent side)

So the following pseudo code shows the consequences:

```
context beginTransaction.
psObject := MyMappedClass newPersistent.
psObject toNRelationship add: anotherObject1.
psObject to1RelationshipParentSide: anotherObject2.
psObject to1RelationshipFKDependentSide: anotherObject3.
context commitTransaction."-> INSERT psObject"
psObject basicToNRelationship isLoaded
```

"→ true with current release, → false with former releases"

```
psObject basicTo1RelationshipParentSide isLoaded
```

"→ true with current release, → false with former releases"

```
psObject basicTo1RelationshipFKDependentSide isLoaded
```

"→ true with ALL releases"

```
context beginTransaction.
psObject := MyMappedClass where: (#id = 1);"load from DB"
psObject to1RelationshipParentSide: anotherObject.
context commitTransaction. "-> UPDATE anotherObject"
psObject basicTo1RelationshipParentSide isLoaded
```

"→ true with current release, → false with former releases"

The new behavior offers better performance but could theoretically have some lacks in a multi-user environment with concurrent access.

10. Obsoletes

Items that become obsolete between Framework releases are moved to specific subcomponents (applications, packages). These subcomponents are named after the supercomponent from which the items were moved with an "Obsolete" postfix.Thus, in IBM Smalltalk, the component ***MicFwPersistenceRdb*** has a subapplication ***MicFwPersistenceRdbObsolete***.

Items in these components have been replaced by more recent implementations in the current release and should therefore not be used in new code.

Caution: Obsolete-code for ObjectBehavior and Peristence will **not** be loaded by default!

Execute the following to remove obsolete code support: (default)

System setSubsystemType: 'micPFWObsoletes' to: 'off'

Execute the following to activate obsolete code support:

System setSubsystemType: 'micPFWObsoletes' to: 'on'

(Re-)load the persistence framework configuration maps afterwards.

11. POM classes

Some components that have become obsolete due to the above changes have been moved to application ***MicFwPersistenceObsoleteSupport***. This applies mostly to obsolete subclasses of ***MicFwPersistence-Manager***. If you are using any of them, you should consider migrating your persistent object managers to the new *POM configuration feature* See 'POM Configuration Editor' (page 187). The only persistent object manager classes that are currently supported are

- MicFwPersistenceManagerOdbc
- ***MicFwPersistenceManagerOracle***.
- MicFwPersistenceManagerESQL
- The best way to change your POM's class is:
- if you have created your own POM class, change its superclass
- if you have created an instance of one of the obsolete POM classes, go to the stored POM initializer and change the #pomClass method to answer the correct class
- After changing the class, you should configure your POM according to your needs (e.g. use a ***MicFwDb2Odbc*** database description for DB2 schema export).

Installing the Persistence Framework

System Requirements

Operating system

The Persistence Framework is supported on the following operating systems:

- IBM OS/2 Warp
- Microsoft Windows 95/98 or NT

Development environment

The Persistence Framework is supported for IBM Smalltalk or VisualAge/Smalltalk Version 4.0 or higher.

Database systems

The Persistence Framework supports the database management systems (DBMS) for respective operating systems as shown in the following table:

DBMS	OS	Type of access
IBM DB2	OS/2, Windows, MVS (embedded SQL) and all platforms supported by the AbtDatabase-Support for VA Smalltalk	DB2 CLI, embedded SQL
Sybase SQLAnywhere	OS2, Windows	OS2: supplied WODXXX.DLL
Windows ODBC	Databases with ODBC capability (Interbase, MS-SQL Server)	According to availability
ODBC Driver Manager	Oracle 7 and 8	All platforms supported by the AbtOracleDatabase-Support for VA Smalltalk (OS/2, Windows, AIX)
AbtDatabaseSupport (native Oracle OCI)		

Table 2: Supported relational databases for operating system environments

ODBC

To access databases using ODBC, a valid ODBC installation must be present. This is usually created automatically under Windows when the DBMS is installed. Under OS/2, third party manufacturer products may have to be used. However, IBM DB2 and VisualAge/Smalltalk are supplied with ODBC installations, including under OS/2.

The ODBC driver for the target database which is used must support the ODBC Level 1 functions. The following requirements are also made of the driver:

- The SQLColumns, SQLForeignKeys and SQLPrimaryKeys functions are used within the scope of the schema import. If these are not available, the framework provides native schema import components for some databases or you can implement the schema import yourself.
- Support for multiple cursors is required when processing inheritance hierarchies, when working with iterators or when using multi-threading.

Installation overview

IBM Visual Age Smalltalk (version 4.5 recommended) must be already installed on your computer before installing Frameworks.

Installing Frameworks consists of 2 main steps:

- Importing the configuration maps from the library file on the CD ROM (\manager**V50.dat**) to the library file on your computer (typically **\vast\mgr50.dat**).

- Loading the applications in the imported configuration maps (along with any other required maps) to your image file (typically \vast\abt.icx).

Importing configuration maps

Frameworks is installed in your Smalltalk environment by importing the required Frameworks applications using the Visual Age Configuration Maps Browser. The applications are imported by importing the configuration maps that contain the Frameworks applications (applications contain the classes and methods).

1. In the System Transcript: Select ***Tools / Browse Configuration Maps***. The Configuration Maps Browser opens. In the Configuration Maps Browser: Select ***Names / Import*** ("names" refers to the names of confi-gurationmaps).
2. In the dialog "Enter the full path name of the library" ("library" refers to the .dat file that contain the configuration maps): Double-click on the Frameworks library file **V50.dat** (on the CD ROM).
3. Selecting the configuration maps for importing and click ***"OK"***.
4. The configuration maps and their applications are imported to your library.The message "Finished importing from **\vast\V50.dat**" appears in the System Transcript.

Loading application maps (with required maps)

When a configuration map is loaded, all of its required configuration maps are also loaded. Therefore loading the configuration maps in the order described below will minimize the required effort to load all configurationmaps.

5. Load with required maps the *micFrameworksBaseDevelopment*.
6. Applications that support Object Behavior functionality: *micObjectBehaviorFramework Development*.
7. Load with required maps the *micApplicationFramework Development*.
8. Required for support of visual programming in the Composition Editor: *micApplicationInteraction - Platform GUI Enhancements Development*.
9. Required for Drag and Drop support: *micApplicationInteraction - Drag and Drop Runtime*.
10. Required for authorization support: *micApplicationService - Authorization Runtime*.
11. Required for object validation: *micApplicationService - Validation Runtime*.
12. Required for multi-language sup-port: *micApplicationService - Multi Language Development.*
13. Select ***File / Save Image***.

Importing the code with Load Features

14. To import the code into your library and to load it into your image run

```
Tools -> Load/Unload Features
```

From your Transcript window: Select Mynd Persistence Framework and its Examples.

1

Concepts and Architecture

1.1. Introduction

Object orientation has started its triumphant march in the development of end-user software. This applies in particular to applications with graphical user interfaces because of the wide range of objects and possible interactions which are involved. However, as far as data is concerned, the more "classical" forms of storage and processing which work with file systems, hierarchical databases, network databases and, since the 1980s, relational databases still prevail. No software which is geared to company business processes can ignore this long-term data resource.

Mynd has responded to this challenge with a Framework targeted at data integration in Smalltalk applications. The development of a Framework is an ideal solution, since it is often necessary to repeat the same programming work in different projects in order to meet precisely this requirement - access to company data. The inclusion of data stored in databases, especially relational databases, was and is the main focus of development. On the other hand, the databases that have already been introduced in user companies can also be used to accommodate new Smalltalk objects and store them persistently.

The following explanations relate in general to all imaginable combinations of object and data models. However, the main focus is on the relational model, which will be dealt with specifically at various points.

1.1.1. Requirements and Areas of Application

Everyone is talking about object-oriented application development nowadays, and it is already being used intensively in many companies. In contrast, object-oriented databases are hardly used by companies for storing mission-critical data, even though they are extremely suitable for object-oriented programming. This situation may be attributable to the following facts:

- The field of object-oriented databases is still young. You could say that it is still at the research stage.
- There are still no binding design standards. Standard ODMG93, which is intended to standardize the handling of persistent objects, is still very new and is awaiting implementation and validation by the manufacturers.
- The security features and multi-user capability of leading relational database management systems are without equal.
- Expertise and experience in the field of object-oriented databases is still thin on the ground.

These shortcomings are preventing large-scale changes of direction because important data exists for much longer than the programs which process it; no decision-maker can risk endangering this data or reducing processing speed. The master data and operational data, and thus the database management systems (DBMS) which manage it therefore become the main focus of software development. Relational database management systems (RDBMS) have asserted themselves as a de facto standard in many places. They are widely scaleable and support an extensive range of uses because of the many tools which are available (e.g. report generators).

The relational data model was enthusiastically accepted when it was introduced by E.F. Codd in 1970 see [CODD70], and has continued its triumphant march through the corporate data processing scene. In the commercial sector in particular it offers an ideal and simple concept for mass data processing, list generation and analyses. Nevertheless, it still has significant drawbacks:

- In order for applications to be able to work at acceptable speeds with large volumes of data, most of the data models are denormalized to a large extent. This creates a multitude of additional tasks such as mastering redundancies and copying services between databases and applications-specific relationships.
- Only a few scalar data types are supported. This leads to considerable complexity in the applications, since the latter have to laboriously dismantle their objects (that have been assembled repeatedly) into scalar components when they need to be stored and have to install them again in the objects when they are fetched. This has been found to constitute a considerable part of the overall workload in large development projects.;
- Combined objects (aggregates) often have to be distributed over several relationships. This prevents these objects being handled "naturally", since their direct context is lost and the individual parts of an object can often only be located and assembled again using laborious search mechanisms (example: parts list structure).

A Framework which wants to bridge the gap between the object-oriented "program world" and the non-object-oriented "data world" has to take all of this into consideration. In particular, it must take into account the longevity of the data and data structures. This may mean that the same object model has to be represented in different ways in different environments.

The Persistence Framework bridges the gap between relational databases and object-oriented applica-

tions. It is also possible to connect non-relational databases, especially object-oriented databases. A uniform interface prevents the need to change an application's object network. This makes it possible to design applications in a flexible way that does not have to concern itself with the finer points of different database management systems. As a result, the Framework is suitable for almost any application in which objects have to be stored in databases for long periods.

An interface for relational databases currently exists. The Persistence Framework is complaint with the ANSI SQL 89 standard (= ANSI SQL 92 Entry Level), extended by rules for referential integrity (see [CANN]).

Other interfaces are planned or are being developed for customers. The most straightforward facility is provided by an interface to object-oriented databases, that would make it possible to smoothly migrate the application created using the Persistence Framework to the new database technology. The Framework is open for new developments and standards.

1.1.2. Basic Structure and Operation

1.1.2.1. Persistent objects

In the Persistence Framework, persistence is a property that is inherited through specialization from the ***MicFwPersistentObject*** framework class. An application class whose instances can be made persistent at any time must therefore be derived from ***MicFwPersistentObject*** (or one of its subclasses). For example:

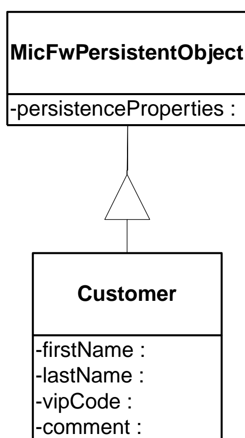


Figure 3: Persistent class derived from MicFwPersistentObject

This provides the customer with the protocol and state which are needed to support framework functionality. Among them are:

- transition between, and maintenance of, persistent and transient states
- loading from storage medium
- deleting from storage medium
- propagating object changes to the persistence medium

Many of these properties and operations are never directly invoked by the application programmer, but rather are used implicitly by the PFW.

1.1.2.2. Interoperation with Other Frameworks

The Object Behavior Framework, with its *Typing*, *Relationships* and *Object Transaction* components, forms the basis of the Persistence Framework. Since the OBFW extends the base Smalltalk object model by these concepts, the PFW can be considered to be extending the OBFW model by the concept of persistence.

The Object Behavior Framework properties that are most important for use of the PFW are briefly described in this section. For details, please refer to the OBFW User's Manual and the PFW programming guide.

1.1.2.2.1. MicFwPersistentObject

The ***MicFwPersistentObject*** class is the abstract base class for all persistent application classes. The most important properties it inherits from its superclass ***MicFwApplicationObject*** are versioning capability (i.e. object transactions), typing capability and relationships.

1.1.2.2.2. MicFwPersistenceContext

The PFW extends the transaction capabilities of the OBFW by providing a type of transaction context whose changes are committed not only to stored objects, but also to the associated persistent storage.

This is implemented by the ***MicFwPersistenceContext*** class, which is derived from ***MicFwTransaction-Context*** (these classes are never actually referenced directly in an application program).

If a persistent transaction context is committed, it passes all changes on to the associated persistent object manager, which then processes the changes and propagates them to the underlying DBMS. In the same manner, an abort to a persistent transaction context is mapped to a rollback operation for the DBMS.

1.1.2.2.3. Typing

The typing capabilities of the OBFW are used to perform necessary type conversions before and after accessing the DBMS.

1.1.2.2.4. Relationships

The relationship semantics as defined in the OBFW are also continued in the PFW. To make a relationship persistent, it must be mapped within the enclosing persistent object class.

If a persistent relationship is resolved for the first time and the relationship is mapped in the currently active POM, the target objects are transparently retrieved from the DBMS by the POM. The loaded objects are then inserted into the relationship and presented to the application program just as expected. The only difference in traversing persistent relationships is increased processing time - depending on the speed of database access and the number of objects referenced.

The PFW also provides the capability to process relationships in a *stream*-like manner, accessing the referenced objects sequentially one by one. This allows large numbers of objects to be processed without loading them all into memory at once.

1.1.2.3. Mapping, typing and relationships

1.1.2.3.1. Mapping

The term *mapping* describes the assignment of database locations to data values from corresponding persistent objects. In relational databases, this is achieved by representing the attributes (instance variables) of a class as columns of a table in a database schema. The Persistence Framework attempts to preserve the Smalltalk object model properties by providing as much flexibility as possible for the actual mappings from the object model to the database model. For example, it is not necessary to store all the instance variables of one class in the columns of a *single* table; sometimes this is not even desired or sensible (for the objects in an instantiated (specialized) class which inherits instance variables from its base class, for example).

1.1.2.3.2. Typing

The term *typing* defines the transformation of certain "scalar" values from the object model to corresponding types in the target database system. To make it easier, it will always be assumed in the following discussion that we are talking about a relational database, which is most often the case in the real world. The ANSI SQL 89 standard, for example, defines a fixed number of scalar data types such as *Integer*, *Float*, *String*, etc. The relational model requires that all stored data must be represented using these data types. In contrast, the Smalltalk object model derives its strength from the availability of an extensive and extensible number of instantiated (specialized) types (classes).

Before values from persistent objects can be stored in their assigned database location, they must therefore often be converted to the data types of the target DBMS. The Framework provides default conversion methods for most common data types and allows extensions for user-defined data types. Upon transferring data between the different worlds, the PFW performs the appropriate conversions to ensure the correct representation of data in the respective environment.

1.1.2.3.3. Relationships

In Smalltalk, relationships between objects are represented as object pointers. No distinction is made between complex and simple (scalar) data types since everything is a full-scale object and, conceptually speaking, there is no difference between a relationship and an object pointer.

However, only scalar data can be stored directly into columns in relational databases. For complex data types, relationships must be represented by resorting to other means such as foreign keys. The Framework supports the representation of a large variety of relationships through the corresponding database construct. The following options exist:

- If an instance variable contains a reference to *one* complex Smalltalk object and you want to store the object in a *single* column of a table, a type converter must be available to convert the Smalltalk object into a data type that is compatible with the SQL data type of the column. The instance variable can then be mapped to the column as usual.
- If an instance variable contains a reference to *one* complex Smalltalk object and it is *not* desired or feasible to store the object in one column of the target table, it is possible to store the object across several columns of the target table by using *aggregate* (or *embedded) mapping.* This means that the object ref-

erenced in the instance variable is stored in the same table as the enclosing object (the object containing the reference). One main characteristic of a relationship mapped as an *aggregation* is that, in terms of the DBMS, the aggregate is identified through the object in which it is embedded.

If none of the above options is applicable or an instance variable contains a reference to a *collection* of complex Smalltalk objects, the relationship must be mapped to a corresponding DBMS construct. The most desirable and common solution is to use foreign key definitions from the underlying database, which can be made known to the Framework either automatically through the schema import facility or declared manually. The PFW will use the foreign keys to resolve relationships and update the corresponding columns when the relationship is changed.

In order to maintain relationship and type declarations and to ensure integrity and validity within the Smalltalk model, the PFW makes use of Object Behavior Framework facilities.

1.1.2.4. The persistent object manager

One objective of the PFW was to keep different components of the architecture apart, so that it would be possible to combine them in a flexible manner at a later point in time. The decision was therefore made to separate the mapping information for a class from the actual Smalltalk implementation of the class by storing it in associated "class mapping objects". These mapping objects reside inside a persistent object manager (POM) and govern the database access for the respective classes. A persistent object manager thus manages the mappings for a fixed set of Smalltalk classes. This design offers several advantages:

- Different mappings can be defined for the same class within different persistent object managers. For example, a customer may be stored in 2 tables in one database and 1 table in another database. It is even possible, for example, to map the same class into a relational database and an ODBMS using different persistent object managers.
- Persistent object managers, and therefore mappings, can be exchanged easily and flexibly whenever needed. This can be done both at runtime and at development time.
- The persistent object manager has the following responsibilities:
- store and process mapping information for persistent classes
- maintain persistent transaction contexts
- maintain the object identity cache
- define a configuration for database access

A persistent object manager is an instance of any concrete subclass of abstract class ***MicFwPersistenceManager***. The subclasses implement the mapping for different target database models (e.g. object-oriented, relational) and any additional properties.

1.1.2.4.1. Storing and processing mapping information

For each class that has been mapped within a POM, the POM contains an object that describes the mapping. This description object is accessed whenever an object is stored in or retrieved from the database. The exact mapping properties depend upon the models between which the mapping is done. For example, if the POM maps between Smalltalk and an OO database, the mapping will most likely be a simple correspondence between the Smalltalk class and a class in the domain of the OODBMS. In the case of a relational DBMS, the mappings are much more complicated.

Whenever the persistent representation of an object is accessed (e.g. loading, committing changes), the access is processed by the POM. Based on the mappings found for the associated class, the corresponding DBMS calls are generated and performed (e.g. SQL for relational DBMS).

At program runtime, every persistent object is associated with a POM. Upon resolving a relationship, the persistence operation for loading the referenced objects is also routed through the POM.

1.1.2.4.2. Maintaining persistent transaction contexts

If the transaction interface is used (see ‘Transaction interface’ (page 37)), the persistent object manager is responsible for the administration of persistent transaction contexts. A context can be obtained from the POM by sending the #newPersistenceContext message. All changes occurring while this context is running are associated with the POM and will be processed through the POM when committed.

1.1.2.4.3. The object cache

If the transaction interface is used, objects are registered in an object cache within the associated POM. The lookup in that cache takes place based upon the key values defined by the object class mappings. This ensures object identity when persistent objects are loaded from the DBMS.

1.1.2.4.4. Persistent object manager configuration

A POM class may itself consist of different components that can be combined in various ways. Depending on the concrete persistent object manager class (and the underlying database model), different configurations may apply. For example, the items that can be configured for relational persistent object managers

(subclasses of ***MicFwPersistenceManagerRdb***) are:

name	task	description
Database Description	describes the data types of the underlying DBMS	A static component that holds the data types and their properties supported by the DBMS-API. Also used for SQL-dialect specifc behavior during SQL generation.
Statement Executor	call Database API	Configures the interface that executes the generated SQL.
Commit Handler	handle object transactions	This defines the commit strategies such as optimistic, pessimistic.

Table 3: Configurable items for relational persistent object managers

1.1.2.4.5. Persistent object manager class hierarchy

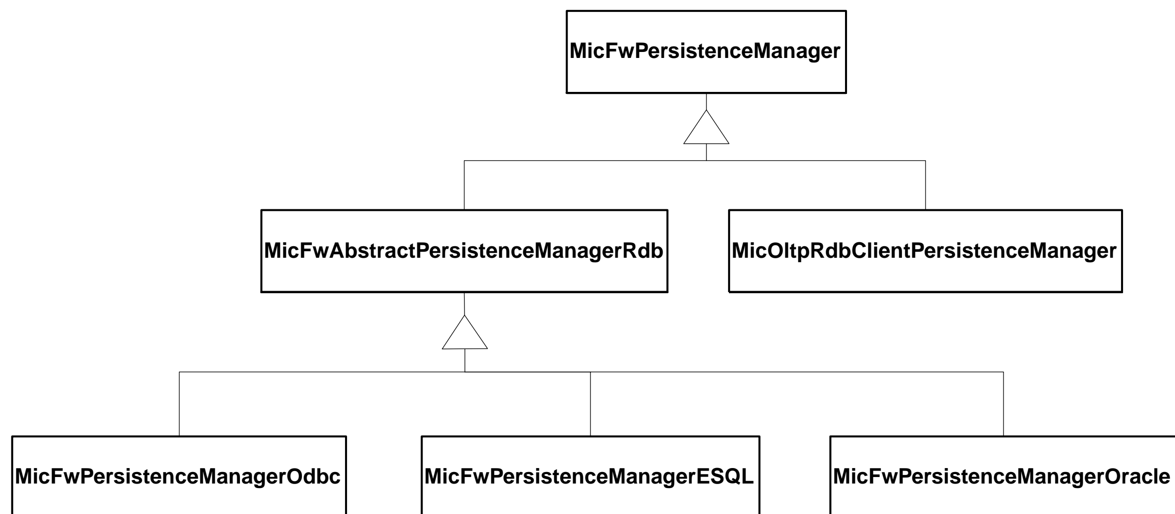


Figure 4: Persistence Manager Hierarchy

1.1.2.5. Application programming interface

In order to work with persistent objects in a program, the objects must first be loaded from the persistence medium into the Smalltalk image. Part 1 of this chapter describes the different interfaces provided by the PFW for this purpose.

Within the program, persistent objects will be manipulated in various ways. Part 2 explains how the changes that occur within the object model are propagated to the persistence medium. For details on these concepts, please refer to ‘Programming Reference’ (page 91).

The PFW was designed with the intention of allowing persistence to appear as a natural extension to the Smalltalk programming model. The number of additional APIs was therefore kept to a minimum. A great deal of persistent behavior is provided implicitly.

In particular, the intention was to avoid the need for calls like the following

```
MyObject store.
MyObject update.
```

In order to allow this, the PFW is tightly integrated with the Transaction features of the Object Behavior Framework.

Persistent functionality is basically provided by the ***MicFwPersistentObject*** and ***MicFwPersistenceManagerRdb*** classes and their subclasses.

The protocol of ***MicFwPersistenceManager*** and its subclasses is used for tasks such as:

- obtaining a database connection
- starting a transaction context for persistent operations
- newly loading objects
- executing direct persistence calls
- ***MicFwPersistentObject*** provides a protocol for:
- maintaining the persistent state of an object (newPersistent, delete, refresh)

1.1.2.5.1. Loading persistent objects

The normal course of actions within a program will start with the loading of some persistent object from the DBMS. Often, this is a root object from which relationships are traversed in order to retrieve associated objects. These objects are then manipulated, and the results are stored in the DBMS. Loading can involve either one specific object or a collection of objects that satisfy a certain condition.

A persistent object identifier (class ***MicFwPsObjectIdentifier***) is used to load a single instance of a persistent class. The persistent object identifier is supplied with the values needed to identify the object, and the object is loaded.

A persistent extent identifier (class ***MicFwPsExtentIdentifier*** or ***MicFwPsExtentIdentifierSQL***) is used to load a collection of objects. It is supplied with the condition to describe the extent, and then loaded. The condition can be formulated either in object query language (see ‘Object Queries’ (page 133)) or by using a native interface (e.g. SQL for relational databases).

Objects that are referenced through relationships from other objects that are already loaded are loaded on demand as soon as the relationship is referenced in a Smalltalk statement. For n-ary relationships, this can mean that a large number of objects may be loaded into memory. If that is not desired, a relationship can be accessed using an iterator. This allows the application to load and process each object in the relationship individually. For example:

```
all := customer accounts asOrderedCollection."all Accounts referenced by the
customer are loaded immediately"
customer accounts asIterator untilEndDo: [:each |
self process: each].
"each Account is loaded and processed individually"
```

As mentioned previously, the condition for a persistent extent identifier can be specified in object query language. The query language defined by the PFW is modeled on the object query language as defined in the ODMG93 standard for object oriented databases. The language has been implemented in a way that makes its usage within the Smalltalk environment easy and intuitive. However, some special rules apply which are documented in (see ‘Object Queries’ (page 133)). An example query would be:

```
pom getInstancesOf: Account where: [:accnt |
(accnt fullAmount > 100 and: (accnt type = 'xyz')].
```

1.1.2.5.2. Modifying persistent objects

Persistent objects are modified in three ways:

- creating an object
- changing instance variable values (including relationships)
- deleting objects

The PFW was designed with the intention of making all these operations as natural to the Smalltalk environment as possible. For this purpose, the PFW provides an interface that integrates seamlessly with the Smalltalk programming model. If ease of use is not the main priority, other interfaces are available with corresponding advantages/drawbacks. The interfaces are:

- transaction interface
- direct interface
- native (SQL) interface

1.1.2.5.2.1. Transaction interface

The transaction ***interface*** is located at the highest level of abstraction. It requires that all operations are carried out within the limits of a framework transaction context. See chapter Interoperation with Other Frameworks: Interoperation with Other Frameworks. Persistent objects are created and modified just as if they were plain in-memory Smalltalk objects.

The changes are propagated to the persistence medium as soon as the transaction context is committed.

Object identity is maintained through an object cache within the persistent object manager. The DBMS transaction is managed implicitly by the framework transaction. An example:

```
context := pom newPersistenceContext.
context beginTransaction.
customer := Customer newPersistent."object creation"
customer name: 'George Washington'."object modification"
context commitTransaction."changes are written to the DBMS and committed"
```

1.1.2.5.2.2. Direct interface

The ***direct interface*** deviates more from the Smalltalk programming model in that it requires all persistence operations to be explicitly issued to a persistent object manager.

A transaction context is not required, and changes are written to the DBMS immediately. Object caching is

optional when using direct interface methods to load objects. The DBMS transaction is managed by the application.

Update operations to already stored objects are not supported in the direct interface at present.

Example

```
customer := Customer new."Smalltalk object creation"
pom insertObject: customer."store to DBMS"
pom deleteObject: anotherCustomer."delete from DBMS"
pom commitDatabaseTransaction."commit to DBMS"
```

1.1.2.5.2.3. Native interface (SQL interface)

The ***native interface*** gives direct access to the underlying storage system. In the case of an RDBMS, this interface allows you to issue SQL statements. Synchronization with in-memory objects is not performed.

Example

```
pom executeSQL: 'DELETE FROM CUSTOMER'.
```

1.1.3. Some examples

1.1.3.1. Transaction interface

```
pom connectWithDriverPrompt."ODBC connect"

context := pom newPersistenceContext.
context beginTransaction.
aCustomer := Customer newPersistent.
aCustomer
   firstname: 'Joe';
   lastname: 'Doe'.
   aCar := Car newPersistent.
aCar
   brand: 'Ford';
   model: 'Escort';
   type: 3.
aContract := Contract newPersistent.
aContract
   customer: aCustomer;
   car: aCar;
   rate: 100.
context commitTransaction
```

1.1.3.2. Direct interface

```
aCustomer := Customer new.
aCustomer
   firstname: 'Joe';
   lastname: 'Doe'.
   pom insertObject: aCustomer.
```

1.1.3.3. Executing SQL

```
statementDescriptor :=
pom executeSQLForFetch: 'select * from Customer where id > 20'.
statementetDescriptor nextRow inspect.
statementDescriptor close.
```

1.2. Static and dynamic POM components

This chapter covers the current POM architecture in more details than the previous chapters. To understand this issues, it is probably neccessary to be familar with the contents of 'Mapping Guide and Reference' (page 71) and 'Programming Reference' (page 91).

Since R3.4 V1.1 the POM-architecture and mapping classes have been redesigned to fulfill the following requirements:

- Reusage of POM-components in different POMs (support for client specific state in different POMs)
- Reentrancy

1.2.1. Design changes to former releases:

A POM instance has several responsibilities, as there

- provide API methods for the application programs to use
- hold mapping information of classes to database tables
- interact with other framework components, e.g. relationships and image transactions
- hold dynamic state information, e.g. object identity cache and database connection

Some of these responsibilities require, that the POM is an object with dynamic state, e.g. state that changes its contents while the POM is doing some work. A POM instance is quite a big object and most of its size consists of the mappings, which have an absolutely static manner. So the redesign maxime was to separate static components such as the mappings from dynamic components. Since this has been done, it is possible to share the static components between several POMs so that the dynamic information can be used ST-process specific or client specific as this will be demonstrated in the examples below.

Algorithms that work on the static components and tasks performed by static POM components have to be reentrant so all POMs that share the same static components can perform their tasks multi-threaded without causing unwanted side effects.

1.2.2. Dynamic POM components

The POM itself still is the object that the application programmer is working with and its responsibities are the same as mentioned above. So the POM itself is a dynamic object which state changes when you use it, e.g.

When you connect a POM with the #connectTo:user:password: method, the POM connects to a specific data source and holds a specific connection handle.

Other dynamic components of the POM are stored in the POMs instance variables:

#eventTable, #instanceName, #persistenceContexts, #cache, #commitHandlerClass and #statementExecutor

This means, that a POM is always associated with one Identity Cache, one statement executor and is configurated to use a specific kind of commit handler.

`#statementExecutor`

The statement executor is responsible to communicate with the underlying database API and performs all SQL statements. In contrast to former releases, the statement executor has NO backreference to the POM that it is used in.

This means, that it is possible to assign the same (identical) statement executor to several POMs. This allows several POMs to use the same database connection.

To assign a statement executor to a POM, you can use the method:

```
<pom> setStatementExecutor: <aStatementExecutor>.
```

Pay attention that the assigned statement executor is compatible with the POM! (So do not assign an ESQLStatementExecutor to an ODBC POM!)

1.2.3. Static POM components

The POM has an instance variable named #core which holds an instance of **MicFwPomRdbCore**.

The POM-core is the root object for all static POM components as there are:

- all mappings
- the registry with registered statements, queries etc.
- database description and specific configurations

There are separate POM-core classes for:

- Runtime-POMs (RT-POMs). The core holds runtime relevant information only.
- common "tool-POM"s. Here the core holds some additional development information.
- OLTP client POMs. The core holds and handles mapping information that come from the server POM that the client POM is connected to.

Note: No static component must have a reference to a dynamic component!

This means, that no mapping references the POM instance (like this was in former releases), because the POM itself is dynamic and the POM-core can be used by several POM instances. Instance variables in the static components that referenced the POM in former releases, are either deleted in the current release or contain a reference to the POM-core.

The static components have the following functionality/responsibility:

- hold all mapping information
- manage registriered statements and queries
- generate most of the SQL statements

No dynamic object is neccessary to perform these tasks.

Some other tasks require that a dynamic object is available. This dynamic object (e.g. the POM or a PersistentObject etc.) will always be handed over to the static component as a parameter.

Such tasks that require both a static and a dynamic component are:

- Compute parameter values for SQL statements
- Execute SQL statements (this needs a statement executor)
- Object caching or object resolving (this needs the object identity cache, held by the POM)

The following picture gives an overview of POM-components. Dynamic components are orange, static components are grey and blue.

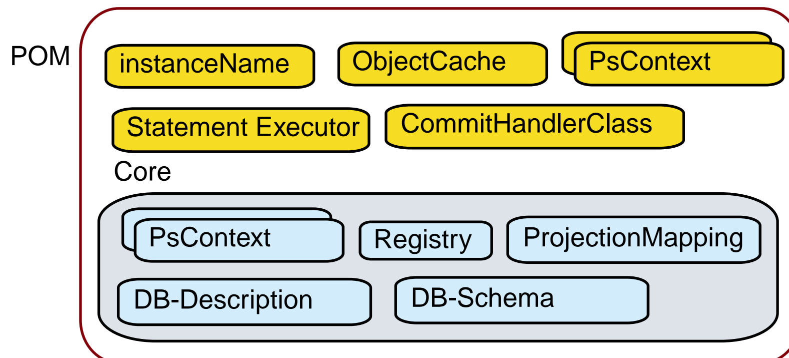


Figure 5: Static and dynamic POM components

1.2.4. Sharing a POM-core between different POMs

Lets assume that you have a POM-instance and some process needs to perform a persistence action with its own object cache.

In former releases you had the following possibilities to solve this problem:

You created a second POM for this process.

The second POM had to be created during development time. The second POM had its own object cache and probably exactly the same mappings. The disadvantage of this solution is that the second POM does not reuse the mappings of the first POM, even if they are the same. This solution is quite inflexible and needs a lot of memory.

Exchange the POMs cache while the process is running

When the process runs in a forked block, it would be very diffuicult to ensure that the cache is always the correct one for each thread running. So this is not multi-threading enabled or very error prone.

In the current release, you could solve this problem the following way:

Create a second POM that has its own object cache but shares its other components with the first POM. So each process can use its own POM. Each POM uses a minimal amount of extra memory.

During runtime, you can create another POM when you send the message #minimalCopy to a POM-instance. The result is a new POM-instance that shares most of the state of the first POM.

Here is the method and its comment:

```
MicFwPersistenceManager>>minimalCopy
"Answer a new instance of the receiver, that shares most of the receiver's
state, such as:
```

```
- core
- configuration
ready to use.
```

The following is also shared (by default) but may be exchanged to allow instance-specific state:

```
- instanceName
- statementExecutor (DB connection)
- cache"
```

So a solution for the example mentioned above could look like:

```
secondPOM := firstPOM minimalCopy.
"give the secondPOM its own cache"
secondPOM initializeCache.
```

As the result, 'firstPOM' and 'secondPOM' share the same core, statement executor but 'secondPOM' has its own object cache. You can give 'secondPOM' its own instanceName:

```
secondPOM instanceName: secondPOM instanceName, '_2'.
```

and put it into the global storage so that you can access it with <pomClass> named: <instanceName>:
secondPOM storeGlobally.

1.3. Mapping Objects to Relational Data

1.3.1. Introduction

This chapter deals with the fundamental aspects of mapping Smalltalk objects to a relational database. For this purpose, the following items are discussed:

- fundamental properties of the relational model
- fundamental properties of the Smalltalk object model
- Mapping - bridging the differences between the models
- Starting an OO project with a relational database

1.4. The Relational Data Model

Note: This section may be skipped by readers who already have a fundamental knowledge of relational databases and database management systems.

The relational model was developed starting in the early 1970's. Today, most database systems on the marketplace are, or claim to be, relational. The relational model is based on a rather well defined theoretical base, which originates in the writings of E.F.Codd and others.

Even though much has been written on the subject of what precisely constitutes a relational database system and what comprises the relational model, there remains some discussion over this subject, most of it academic in nature. Some of the most important documents that define the relational model are [CODD82] [CODD85].

1.4.1. Elements of the relational model

For the sake of simplicity in this discussion, the following items are considered to be integral parts of the relational model:

- data is organized in tables and columns
- columns contain simple values
- tables have primary keys
- relationships between tables are implemented by foreign keys

Data stored in one table may also be loosely referred to as belonging to an "entity" (the term *table* refers to the structural aspect, *entity* to the logical aspects.

1.4.1.1. Table

A table is defined by a fixed number of named columns with associated type declarations. A table can contain an arbitrary number of rows. Rows within tables and columns within rows are not ordered.

1.4.1.2. Primary key

The primary key is the set of columns from a table that together uniquely identify each row in the table. Thus no two rows in a table are allowed to have the same combination of values for the primary key columns. Primary key columns may not contain NULL values

1.4.1.3. Foreign key

A relationship between two tables is implemented by a foreign key. A foreign key is a set of columns whose sequence and definition match that of the primary key of another table. A relationship between two rows (entities) exists if the values in the foreign key exactly match those in the primary key.

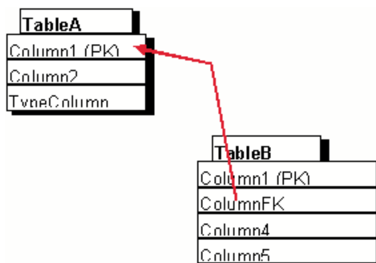


Figure 6: Relational Tables with Primary and Foreign key

1.4.2. Normalization

The term normalization describes a set of principles that are applied to database design in order to obtain certain theoretically (but not always practically) desirable properties of the resulting data model. The basic idea underlying normal forms is to avoid redundancies ("one fact in one place")

Much has been written about this subject and we do not intend to enlarge that volume with new theories. One recommended source for further information is [DATE]. However, we want to give the following short definitions of the first three normal forms (NF), which are the most significant for everyday practice:

- A relation is in 1NF if all underlying domains (i.e. columns) contain atomic values only
- A relation is in 2NF if it is in 1NF *and* every non-key attribute is fully dependent on the primary key
- A relation is in 3NF if it is in 2NF *and* every non-key attribute is non-transitively dependent on the primary key

From this, it follows that every table in a relational database is in 1NF by definition (the presence of a primary key is implicit).

Designing a database in 3NF is desirable because redundancies are avoided and therefore ease of maintenance and robustness are achieved. However, it also causes entities and relationships to be split up over several tables and thus may adversely affect performance. It is therefore not uncommon to *denormalize* a 3NF data model in order to obtain better performance.

By nature, a Smalltalk object model obeys the principles of normalization very closely, since the "one fact in one place" rule is also one of the basic principles of object-oriented design. Therefore a Smalltalk object model maps quite easily to a highly denormalized data model.

The PFW supports different levels of denormalization in order to satisfy performance requirements without sacrificing the object model properties.

1.4.3. SQL

The SQL language (Structured Query Language, see [CANN]) is used by virtually all relational DBMS for accessing data in relational databases.

SQL consists of 2 components:

Data Definition Language (DDL) is used to describe the schema of the database

Data Manipulation Language (DML) is used to manipulate (retrieve, insert, delete and update) data in the database.

An example SQL DDL statement for creating a table with 3 columns would be:

```
CREATE TABLE CUSTOMER (
IDINTEGER NOT NULL,
NAME VARCHAR(130),
CODECHAR(2),
PRIMARY KEY (ID)
)
```

An example SQL DML statement to insert a row into the previously created table would be:

```
INSERT INTO CUSTOMER VALUES (1, 'George Washington', 'XY')
```

An example SQL DML statement to retrieve the previously inserted row would be:

```
SELECT * FROM CUSTOMER WHERE ID = 1
```

All retrieval statements in SQL return data in the form of a "result table" that can be regarded (and is referred to within SQL) as a transient table. The collection of all rows returned in the result table is sometimes called the "answer set".

1.4.4. Foreign key relationship properties

The following describes some terms that are used in connection with foreign key relationships. The entity holding the foreign key is called the child entity, and the one whose primary key is referenced by the foreign key is called the parent entity.

1.4.4.1. Identifying relationship

An identifying relationship exists if the primary key of an entity also contains a foreign key. Depending on whether the primary key has additional columns beyond the foreign key, the relationship is fully or partially identifying. Since the foreign key is part of the primary key, it has NOT NULL attributes.

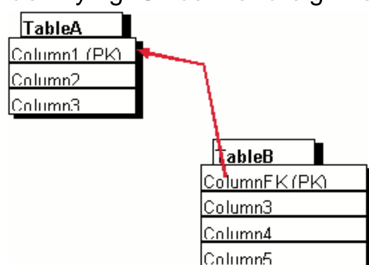


Figure 7: Identifying Relationship

1.4.4.2. Existence dependency

A child entity is dependent upon the existence of the parent entity if the foreign key has NOT NULL attributes. Thus an identifying relationship always incurs existence dependency, but not vice versa.

1.4.4.3. Delete and update rules

Most relational DBMS also support rules for foreign keys that apply whenever the parent entity of a relationship is deleted or its primary key is updated. Some common rules that are available (depending on the

actual DBMS) are:

- ***CASCADE***. The operation on the parent entity is propagated (cascaded) to all child entities
- ***SET NULL***. When deleting a parent entity, the foreign keys of all child entities are set to NULL.
- ***RESTRICT***. If a child entity exists, the operation is rejected. An Integrity Violation Error is issued by the DBMS

1.4.5. Some ramifications

The relational model was defined with 2 main goals in mind:

- to provide a uniform, conclusive theory for data storage
- to define structures that can be flexibly accessed in various ways (e.g. by reporting tools)

However, the conceptional clarity of the relational model does not come without a price. For example, the relational model is not well suited for representing highly complex structures of interrelated entities. This results in possible performance deficiencies and program complexity. The complexity involved in handling even simple structures can be demonstrated using the following example:

Let us say we want to store customers together with their addresses. Every customer can have more than one address. This specification has the consequence that *Address* must be modeled as a separate entity, because it is not possible to store more than one address per customer in the customer table.

In the reverse direction (address -> customer) we will discuss the following options:

- only one customer can reside at an address
- several customers may reside at the same address

Accordingly, additional steps must be taken to implement the relationship.

1.4.5.1. 1-to-N relationship

In the case of a 1:N relationship, Address does not need its own key. Since every address is assigned to exactly one customer, it is already identified through the customer. A foreign key (identifying in this case) is used to reference the customer:

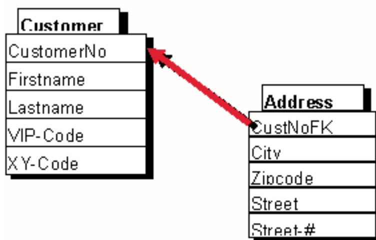


Figure 8: Foreign Key Relationship

The addresses of a given customer with customer number 22 can then be determined using the following SQL query

```
select A.Street, A.Zipcode, A.City
fromAddress  A
whereA.CustNoFK = 22
```

1.4.5.2. N-to-M relationship

The relationship between customers and addresses in the above example can be modeled as both a 1:N and an N:M relationship - with different effects on the underlying data model. To represent an N:M relationship, an associative table with information connecting the rows of data between the tables must be created:

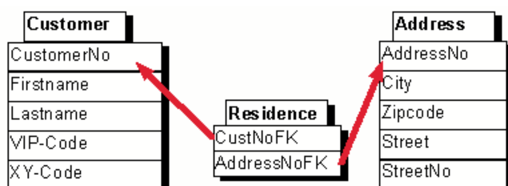


Figure 9: N:M relationship using an associative relation

An SQL query that will return all addresses for the customer with the customer number 22 must then be stated as:

```
select A.AddressNo, A.Street, A.StreetNo, A.Zipcode, A.City
from Address A, Residence R, Customer C
where A.AddressNo = R.AddressNoFK
```

```
and R.CustNoFK = C.CustomerNo
and C.CustomerNo = 22
```

1.5. The Object Model

1.5.1. Object model in general

The object model has evolved since the 1960s. Although literature about the subject abounds, the definition is not as mathematically detailed as that for the relational model. For further study, the reader is referred to [Booch95]. However, there exists little controversy about the basic features of the object model, which are:

Abstraction. Concepts from the problem domain are represented by classes. A class defines the common properties (state and operations) of a set of objects. From a class, objects (instances) are created which share the properties defined by the class. Each object has its own state and identity.

Encapsulation. The state (data) inside an object can only be accessed by invoking operations defined for the object

Inheritance. A class can be defined as a subclass of another class, thus inheriting all properties defined by the superclass. The subclass can add properties and change inherited ones.

Polymorphism. The behavior of objects is determined at runtime, according to the actual object properties. Another term used is "Late Binding".

These descriptions are necessarily very brief and do not claim to be exhaustive.

1.5.2. The Smalltalk object model

The Smalltalk language was one of the first implementations of the object-oriented paradigm in terms of a programming language. Since it represents the basis of the PFW, we will outline a few properties here.

In comparison with the relational model, the Smalltalk object-oriented model is characterized by the following points:

- A higher degree of similarity to the real world through the identification of classes that closely model the entities of the problem domain.
- The concept of instantiation (inheritance) of classes of objects.
- Implicit object identity (through memory address)
- No explicit relationships, no rules for referential integrity
- Dynamic typing. No explicit type declarations are given for variables used in a program

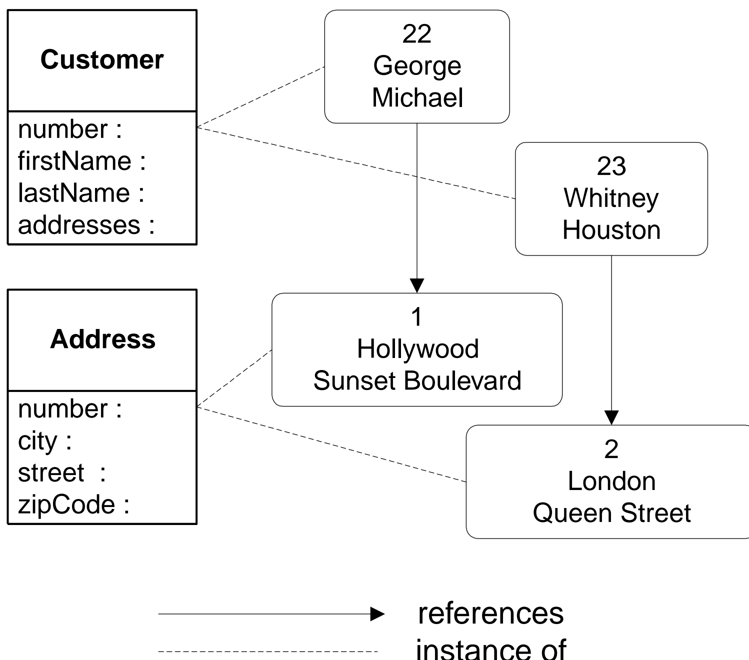


Figure 10: Object Model

1.6. From Relations to Objects (and vice versa)

1.6.1. The challenge

The following main differences between the relational and the object-oriented model of data representation become obvious from the previous descriptions:

- Static typing. Type declarations are given for all attributes in the relational model. These declarations are enforced by the DBMS.
- Object identity. In the relational model, entity identity is defined by a set of attributes that together form the primary key of the entity. Sometimes these attributes are taken directly from the application domain (e.g. name), but these key attributes are more often artificially generated (surrogate keys) in order to ensure uniqueness. In the object model, identity is implicit (object memory address)
- Relationships. In the relational model, relationships are represented by columns in one entity that reference the key of another entity. N-to-M relationships are implemented by means of associative tables. Referential integrity is maintained by the DBMS. The object model uses simple object pointers and object collections for relationships. Referential integrity is not maintained in the object model. Therefore, the relational model can be regarded as more advanced in this area.
- No inheritance. The relational model does not support inheritance directly
- These differences incur a significant overhead when transforming data from an object-oriented representation to a relational one. This would normally have to be carried out by the application program.
- Due to the strong proliferation of both models in today's data processing environments and the complexity of the task, Mynd decided to implement a framework of Smalltalk components (PFW) which would shield the programmer from these complexities and provide a flexible and easy-to-use interface for storing Smalltalk objects in relational databases.
- The aim is to give the programmer a tool for declaring the corresponding structures on both sides (the *mappings*). At program runtime, the Framework would use these declarations to transparently store or retrieve objects to/from the database.
- A small API (Application Programming Interface) should be defined by the Framework for persistent objects to manage persistence related properties. In general, however, persistent objects should behave just like other Smalltalk objects. In particular, the Framework should transparently maintain relationships by implementing referential integrity in the object model and encapsulating the persistent representation of relationships.

1.6.2. Mapping

To perform the transformation between relations and objects, the PFW needs rules that describe the corresponding items on both sides. The items from both models are:

relational			
Table	Column	Primary Key	Foreign Key

Table 4: Description of relations

object-oriented				
Class	Subclass	Object	Instance Variable	Collection

Table 5: Description of objects

The transformation rules between these items are called mappings. There are different types of mappings, and in many cases there are several alternative mappings between the two sides. These are described in 'Mapping Guide and Reference' (page 71).

1.6.3. Mapping Tool

The PFW provides the STOPF tool for performing the declarative part of the job. It is used to carry out the following tasks:

- STOPF is used to interactively define the mapping between an object model and the corresponding DB schema.

- An existing DB schema can be imported from a database.
- The DB schema can be defined or manipulated manually (e.g. foreign key declarations can be added if not present in the physical DB).
- To propagate schema definitions/changes to the physical database, DDL export is accessible from within STOPF.
- Finally, a ready-mapped class can be generated from a table

1.6.4. Other tools

In order to enhance the usage of the PFW, the following additional tools are provided:

- Interactive SQL facility. With this simple tool, SQL commands and Smalltalk expressions can be issued by typing them into a text field and executing them. This is useful for testing and debugging as well as for DDL execution.
- POM generator. A ready-mapped persistent object manager and DB schema can be generated from a set of Smalltalk classes. The mappings that are generated are based on generic assumptions, and therefore the resulting DB schema will display a high degree of normalization. It may, however, be a good starting point if no existing database has to be taken into consideration.
- Embedded-SQL Tools to generate ESQL-Smalltalk code that contains all registered and pregenerated SQL statements of a POM.
- Runtime-POM generator that generates a POM optimized for high performance with less memory usage from a tool-POM.

1.7. Starting an OO/RDBMS Project

Real-world projects most often are faced with conditions that are anything but ideal. This chapter will therefore provide a short guide on how to introduce the PFW into different types of projects. The following general categories will be discussed:

- A relational database already exists. An object model must be developed and mapped to the database
- The object model exists. An appropriate relational database model must be derived to store the object data
- Both relational model and OO model already exist and must be mapped together

We will also make some remarks about different job assignments in projects using the PFW. These are tips that result from experience in existing projects.

1.7.1. Existing data model, new object model

An existing data model for a company is often the starting point for developing business-oriented application software. Before discussing the details of defining an OO model that is to be mapped to an existing relational model, it should be stated that both models are normally developed from the same business domain requirements and can therefore be expected to display a reasonable amount of commonality. Even though the modeling constructs differ, the underlying business concepts are the same.

It is the aim of the PFW to provide ample flexibility in mapping an OO model to a relational database. However, in many cases there can be a significant number of alternatives for solving a certain problem. In other cases, restrictions have to be taken into consideration.

Given the above, there are 2 possible approaches:

- Generate Smalltalk classes from database tables using PFW Tools. This is the easiest solution and requires virtually no expertise. However, the resulting OO model will lack many basic OO properties. This approach is therefore not recommended, especially for larger projects.
- Use the available PFW features to create an OO model that includes as many OO features as possible and is still mappable to the relational model. This approach requires more expertise but should be worthwhile for larger projects. In order to find the optimal solution, a sound understanding of the PFW properties and features is required.

1.7.2. Existing object model, new data model

This situation is similar to the one previously described, in that it can be handled by generating simple default mappings or by manually creating mappings which are customized, but more optimal. The PFW provides tool support to generate a ready-mapped data model (in the form of SQL DDL) from a given object model. The DDL can be executed against the DBMS to create the database tables. Different target DBMS are supported.

Object models tend to be highly normalized. With database models, performance issues have higher priority. This may be counteracted by a high degree of normalization. For this and other reasons, the generated DDL may not be suitable as a basis for the data model. If this is the case, the database should be designed with the object model and the PFW features in mind in order to create a solution that best matches the different requirements. As above, a good understanding of the PFW properties and features is required.

1.7.3. Existing object model and data model

This situation obviously allows the least flexibility. There is no quick solution for non-trivial projects. Instead, a good understanding of the PFW must be used as the basis for bringing the two worlds together.

1.7.4. Activities

Depending on the starting point which is used, the job of mapping can make great demands on the personnel doing the work. For more complex projects, the personnel should have good knowledge of both relational and object-oriented contexts. The quality of this work phase has far-reaching effects on the success of the project.

Past experience in the use of the PFW has shown that it is advisable to summarize the task in hand at this point. This is recommended for development reasons, since there may be a high price to pay for avoiding and resolving conflicts.

The Smalltalk development environments supported by the PFW provide good facilities for making the results of this work phase accessible to all developers. Changes can be adopted by all affected workplaces by simply storing and loading the POM class.

1.7.5. General patterns

For large projects created with the Persistence Framework, refer to the following design patterns to enable easy customization of framework behavior and to avoid problems with newer framework releases.

Subclass the class of PersistenceManager that you are going to use.

Using you own subclass of PersistenceManager[Odbc|Oracle|ESQL] makes it easier to

- Overwrite methods of the superclass
- Add additional behavior to your POM like DDL generation, locking, hard-coded SQL statements, queries etc.
- Change the superclass later on if the frameworks provide a better POM class for your needs or if you want to change the POM class for other reasons.
- Create common superclasses for your domain classes
- Create a common subclass of
  - MicFwPersistentObject (if you do not use the Application Framework) or of
  - MicFwDomainObject (if you are using the Application Framework)

as an abstract superclass for all of the base classes that your application is going to contain. You may also specify a common abstract superclass for all Support-Table-Classes, Read-Only/View-Object classes, IdentifiedObject classes etc. as a subclass of the classes created above.

This allows easy customizations of runtime and development features of the frameworks for your classes by overwriting framework methods and makes the implementation of project-specific standards easy.

1.7.6. Changing framework behavior

Customizing framework behavior and methods should be done with the following rules to avoid compatibility problems with newer releases:

Use methods categorized as “micAPI” only.

Methods categorized as “micAPI” are garantueed to stay compatible in newer framework releases. If a “micAPI” Method becomes obsolete or disappears in newer releases, you will find a migration documentation or migration tool for that release or you may use this method in an “Obsolete”-Support application.

Subclass only methods, that are categorized as “micAPI”.

Refer to the method‘s comment and not to the implementation itself. Call the original super implementation instead of copying-and-pasting the method body to be changed in your subclass.
Remember: The API of a “micAPI” methods stays compatible, but not neccessaryly its implementation.

Methods categorized as “micInternal” may disappear in the next release or have another behavior.

If you think that you absolutly need to send or subclass a “micInternal” method directly in your application, contact us and describe your requirements so we can:

Tell you that this method can be substituted by calling/subclassing another “micAPI” method

Tell you that this method from now on (with the next release) is a “micAPI” method

Keep your requirements in mind, so that we can inform you when your modifications become obsolete or incompatible.

Implement your addtionals/changes in subclasses whereever possible.

Otherwise extent the framework classes in your applications. Only subclass or extent classes, that are “public”, e.g. that will not disappear with the next release.

2

Tutorial

2.1. Introduction

Even the best theory is useless if it can't be applied in practice. This is especially true for software development, because software system users are primarily concerned about having efficient support for their main business objectives rather than the system itself.

This chapter takes a concrete problem to provide a step-by-step description of the fundamentals of developing an application using the Mynd Persistence Framework. The tutorial invites you to experiment and try things out, because the best way to learn is to go through the examples "hands on", modifying the problem on your own, and trying out new ideas that occur to you during the course of the tutorial.

2.2. The example: a customer base

2.2.1. Introduction

Customers are fundamental to a business. The customer is the engine that drives the company towards its goal. In order to operate efficiently in the market, no company can afford not to manage its customer base - regardless of whether it uses electronic data processing (EDP) or not.

Customer bases managed with EDP support are therefore standard practice in medium and large-sized companies. This tutorial specifies and implements a "small" customer base management system. It starts by describing the requirements that the system must meet.

2.2.2. Requirements

Since an RDBMS is already being utilized, the data should be stored in a relational database. The data to be stored relates to the company's customers and employees. The following data is to be gathered for customers and employees alike:

- personal data such as Last Name, First Name and Date of Birth
- the date on which the data was entered and/or the Date of Employment
- one or more addresses complete with Street, Number, ZIP code, City and State

Each customer is assigned to an employee. The customer must also be evaluated using a *vipCode* (important customer, small-business customer, etc.). A *portfolio* may also be put together for each customer that would include the following data:

- savings
- realEstate

The *department* and a *job description* are additional items of information that form part of the employee's data.

The application should support the following functions:

- addition of a new customer together with their main address and the name of the employee assigned to the customer
- addition of a new employee together with their main address
- deletion of a customer or employee
- addition of a new address for a customer or employee
- updating the address of a customer or employee
- deletion of the address of a customer or employee
- updating the name of the employee assigned to a customer

2.2.3. Object and data model

Tip: The commonly used *CompoundWord* notation has been chosen in order to differentiate between class names throughout the chapter. The names of the tables in the database are always written in capital letters in the text, however (example: the *PERSON* table).

The characteristics common to customers and employees alike are to be modeled in a class called *Person*. Since each person can have more than one address, *Address* is a separate class. There is an N:M cardinality association between person and address, i.e. a person can have several addresses and any given address can have several persons residing there.

There is a 1:N cardinality association between employee and customer called *ownedCustomers*. A customer's portfolio is modeled in class *Portfolio*. There is therefore a relationship between customer and portfolio with cardinality 1:1.

The name of a person is modeled in a seperate class *PersonName*. Between a person and his name there is a primitive 1:1 relationship. The Person have a name attribute, the PersonName has no reference to the person.

A specialization of customer, *XmpCustomer*, can have several orders; each order can have several order lines. This is modeled in a N:1 relationship between *XmpCustomer* and *Order*, and in a N:1 relationship between *Order* and *OrderLine*.

An *OrderLine* has S*hipments*, a *Shipment* knows an *OrderLine* (in instance variable *lineItem;* N:1 relationship).

Also an OrderLine has a primitive 1:N relationship to class *Product*. In this example a Product need not to know all OrderLines in which it was ordered.

Each *Product* belongs to a *ProductGroup*. Therefore there is a 1:N relationship between *Product* and *ProductGroup*.
This results in the following class model:

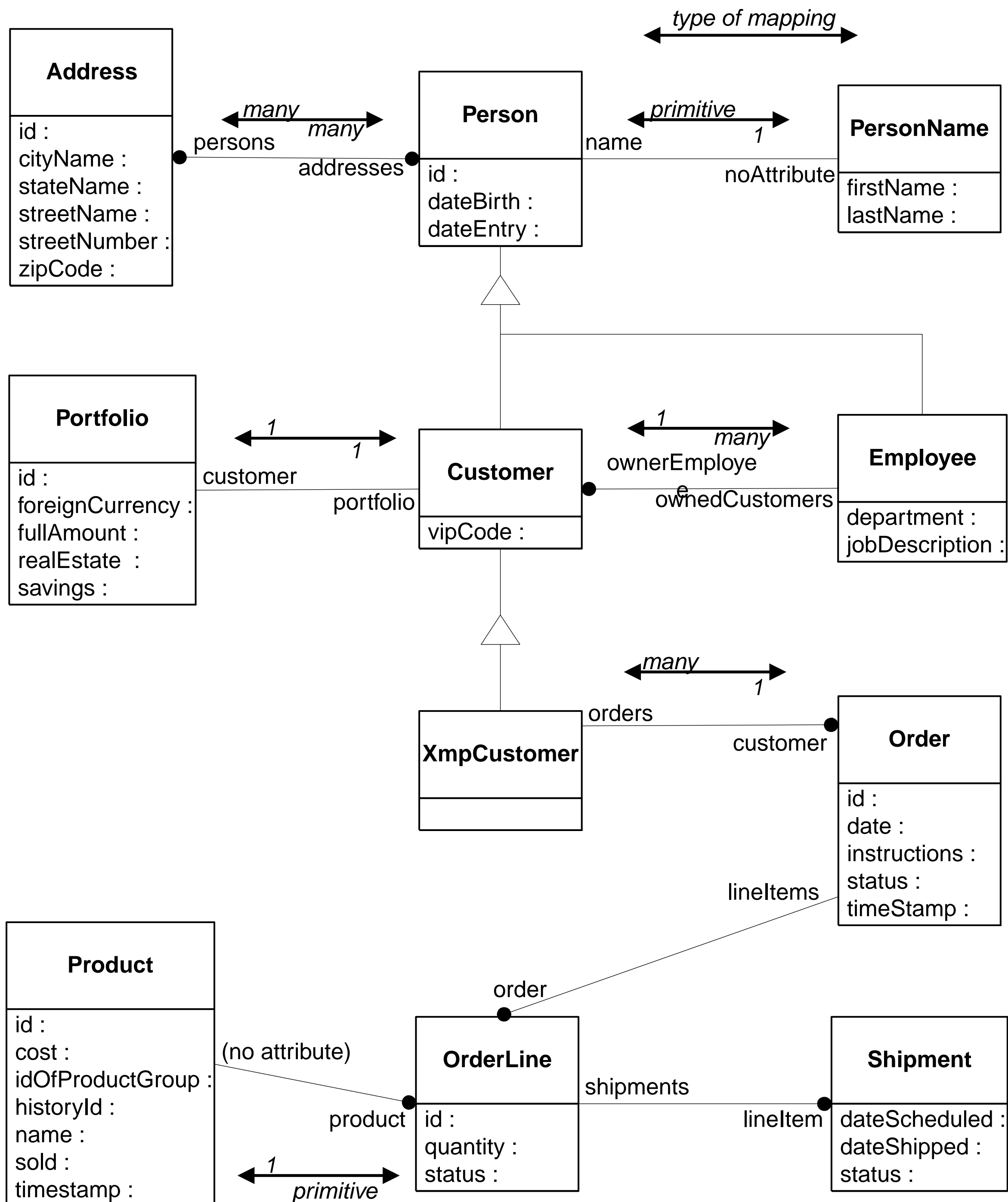


Figure 11: The Customer Base Management class model

The data model roughly reflects this structure. The class hierarchy Person Customer, XmpCustomer, Employee is spread across the three tables PERSON, CUSTOMER and EMPLOYEE. The attributes are spread canonically across table columns with the same or similar names. In order to be able to tell which type of person has been selected during an object retrieval from the PERSON table, *a subType column* is inserted; a unique identifier (the *subType value*) for each of the derived classes is entered into this column.

The relationships for :1 relationships can be found in the table columns that reference other tables using foreign keys. For example, CUSTOMER: the EmployeeId column is inserted into the table as a foreign key reference to the EMPLOYEE table.

For N:M relationships and the :N side of a 1:N relationship, no dedicated table column exists for the relationship. An intermediate table (association table) is generally used for N:M relationships. For example: the RESIDENCE table consists solely of foreign keys to the PERSON and ADDRESS tables, and thereby relationally connects the said tables.

The relationship PRODUCT-PRODUCTGROUP is defined as a NonStandardRelationship; there is no foreignKey reference between these tables.

Note that there are no tables for the classes XmpCustomer and PersonName. The variables lastName and firstName of PersonName are stored in the table PERSON. XmpCustomers are stored in the table CUSTOMER (and table PERSON). Therefore the foreignKey *CustomerId* in table ORDER refers to the CUSTOMER table.

This results in the following relational model:

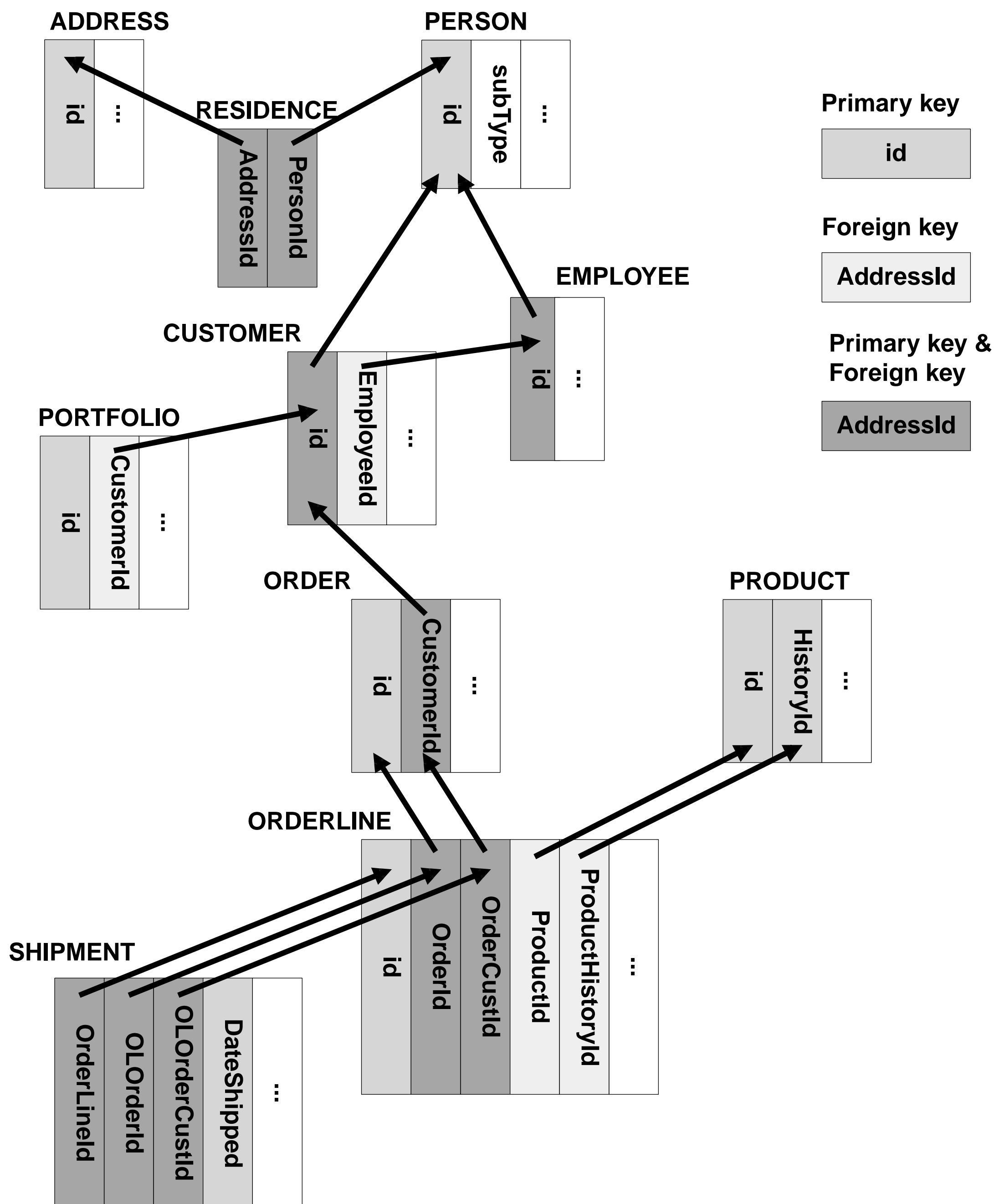


Figure 12: Relational model Customer Base Management

2.3. Object mapping

Tip: The classes used in the tutorial have names beginning with ***MicFwTutorial***... and ***MicFwXmp***.. in the accompanying Smalltalk code segments. In this and following sections this prefix will be assumed and will not be printed, i.e. the class ***MicFwTutorialPerson*** will simply be called *Person*.

Object mapping enables the Mynd Persistence Framework to store, manipulate and reload the application instances - that is to say, at least the application instance variables - to and from a database. Mapping assumes two things:

- that a functioning database is installed and
- that Framework tool STOPF is installed.

This tutorial only describes the operations in STOPF that are actually needed for the example.

A list of the databases supported by the Framework can be found in 'Database systems' (page 27). If you haven't yet installed any of these systems on your computer, we suggest you install the RDBMS *Watcom SQL* for the tutorial.

2.3.1. Preparing the database

The easiest way to create the database schema required for the examples is as follows:

- Select the ***MicFwExamplesPomOdbc*** POM in the Transcript window under *micFrameworks / Browse Persistence Manager*.
- In STOPF, use *Schema / Export* to generate the SQL statements that are needed to create the example schema (subsequently selecting the database type). A workspace is opened containing the DDL file (Database Definition Language) that has just been created. See 'Exporting a database schema' (page 206).
- Execute the generated SQL statements against the database. In order to do this, use the tools that were supplied with your database (e.g. ISQL for Sybase SQLAnywhere), or use the PFW Interactive SQL tool. See 'Interactive SQL' (page 213).

2.3.2. Setting up a POM

In the following discussion, it is assumed that the tutorial database with the complete table schema and the database environment have already been prepared. You may now start your Smalltalk system. When starting for the first time, perform the following steps:

- In the Transcript window, select menu item *micFrameworks* þ *New Persistence Manager...* to create a new Persistence Object Manager (POM). Select ***MicFwPersistenceManagerOdbc*** and click on the *Choose* button. The new, but as yet unnamed, POM is created and ready to be edited.
- Select menu item *Manager* þ *Store Globally As...* in the STOPF dialog window to give the new POM a name which can be used to access it. Enter the name '*FW Examples*' or any other name of your choice in the dialog window which appears. After confirming with *OK*, the name of the POM appears in the title bar of the STOPF dialog window.
- Now exit the STOPF dialog window by selecting *Manager* þ *Exit* and store your Smalltalk image in order to keep your POM. Then continue as follows:
- Select menu item *micFrameworks* þ *Browse Persistence Manager* to select the POM just created from the list of known POMs which appears. Then click on the *Choose* button to edit your POM.
- Select menu item *Manager*þ*Connect to Database...* in the STOPF dialog window. When the database and ODBC environments are correctly configured, a connect dialog window for the database appears. Select a database and confirm with *OK*. STOPF now establishes the database connection. When *'connected'* appears in the title bar, you know that the database connect was successful.
- Select menu item *Schema* þ *Import Schema...* in the STOPF dialog window in order to supply the POM with the database schema. You may select individual tables or all of the tables appearing in the list. Select all tables used in the example and confirm your choice with *Choose*. In the following *Import Options* dialog window use the defaults; just select OK. The tables now appear in the list in the top right corner of the STOPF dialog window.
- You have now prepared your system for initial use with the mapping tool. In order to continue, you now need the application classes whose instances are to be stored in the database. You have the following choices:
- In an *ENVY* environment, load the applications ***MicFwTutorial*** and ***MicFwExamples***.
- Of course, you may also create the application classes yourself. Create the classes specified for the object model described in Object and data model: Object and data model} as classes derived from

MicFwPersistentObject. Define the instance variables according to the data types of the table columns shown in the data model.

After you have obtained a general overview of the main classes used in this context, you are now ready to start with the actual work.

2.3.3. The main STOPF window

The main STOPF dialog window is separated into four sections:

- the left-hand side displays the instance variables of the class currently being edited (*instance variable list*)
- the top right-hand side displays the names of all the tables known to the POM that is being edited (*table list*)
- the names of the tables that refer to the mappings of the class currently being edited are displayed in the upper middle section of the dialog window (*map table list*)
- the bottom right-hand section of the dialog window shows the columns of the table currently selected (*column list*).

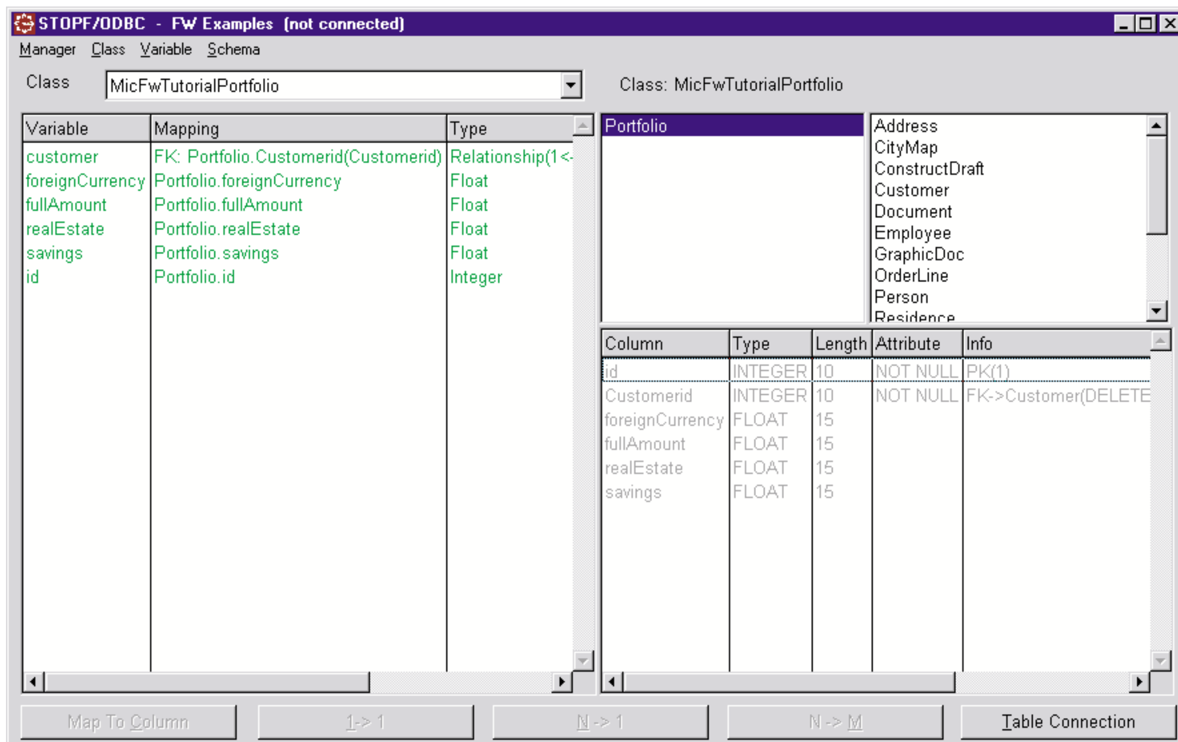


Figure 13: The main STOPF dialog window

Click on a few of the table names in the *table list* and watch how the display in the *column list* changes.

2.3.4. Instance variables

The first and easiest step is to assign scalar instance variables to the respective table columns.

In the STOPF dialog window, select menu item *Class* þ *Choose...* in order to provide the POM with information on the application classes. The list in the dialog box that appears is still empty, because no persistent class has yet been edited. Click on the *Any class...* button anyway and select the *Person* class (or the appropriate class if you have created the classes yourself) from the list of all persistent classes. Confirm your choice by clicking *Choose*.

The instance variables for the chosen class now appear in the *instance variables list*.

Select the scalar instance variables from the *instance variables list* one after the other and the corresponding table columns in the *column list*. The class provided contains the *dateBirth*, *dateEntry* and *id* instance variables.

For example: select instance variable *dateBirth* and table column *dateBirth*.

For each of these choices, activate menu item *Variable* þ *Map to Column*.

Observe how the instance variables list is updated: the mapping and data type information is displayed and the gray color signals that the corresponding instance variable has been mapped. The type information is automatically determined from the type of the database column assigned to it if the type hasn't been

given explicitly.

Something also changes in the *table list* and the *column list*: the table resulting from the class mappings appears in the *map table list*, and the mapped columns are highlighted light gray in the *column list.*

This is right moment to map all scalar instance variables that you find in classes of the Tutorial and Example Applications. These classes are referenced by STOPF dialogs in the next chapters.

2.3.5. Derived classes

The most straightforward way to represent class hierarchies is to spread them across several tables. For each class, there is a table containing the corresponding foreign key references to the table assigned to the superclass. A *subtype discriminator* is used in order to determine the object type that a data record in the base class table represents.

Select the *Person* class.

In the database provided, the *PERSON* table contains the *subType* column. This serves to differentiate between derived classes *Customer, XmpCustomer* and *Employee*.

Select this column from the *column list*. Now activate menu item *Class* þ *Subtyping* þ *Define Subtype Column*.

Select the *Customer* class.

Activate menu item *Class* þ *Subtyping* þ *Define Type Value...*. In the dialog window that appears, you may enter a constant string value for identifying the derived class.

Enter **Cust** and confirm this with *OK*. Notice that the type and length of the value entered are compatible with the type of *personType* column in the *PERSON* table (here: maximum string length = 254).

Proceed in the same manner for the *Employee* and the *XmpCustomer* class. Enter **Emp** and **XmpCust** as the *type value*.

2.3.6. Internal table connection

After you have processed all of the scalar instance variables and have categorized the objects in the *Person*, *Customer*, *XmpCustomer*, *Employee* class hierarchy with a *subtype discriminator*, another element is required in order to be able to work with the said class hierarchy. *Internal relationships* must be created for all classes whose instance variables need to be spread across several tables. This is especially true for class hierarchies. The *Customer* and *Employee* classes, which have been derived from the *Person* class, are examples of this.

- Select the *Customer* class and the *CUSTOMER* table.
- Select menu item *Class* þ *Internal Table Connection...*. In the list that appears, the only way of creating a connection is by using the *Person* foreign key. Select this key and confirm your choice by clicking on *Connect Tables*.
- Proceed in the same way for the *Employee* class.

2.3.7. Foreign key based relationships

Relationships between classes are implemented using the OBFw relationship facilities. In the example, the relationship between persons and their addresses (*Person Address*), between a customer and the employee assigned to that customer (*Customer Employee*), and between a customer and his or her portfolio (*Customer Portfolio*) are examples of this. In each case, both of the classes participating in the relationship were supplied with instance variables for their side of the relationship. Two-sided relationships can be constructed in this way.

In order to make relationships persistent, they must be mapped to some relational counterpart. In this case, the foreign key relationships between the tables which can be seen in STOPF are the starting point for the mapping, i.e. only tables that contain a foreign key are available for relationships.

Now take a look at the 1:1 relationship between *Portfolio* and *Customer*. The *PORTFOLIO* table has a foreign key which references *CUSTOMER*, but not the other way around (the portfolio *depends on the existence* of a customer). Because of this, mapping must start with the *Portfolio* table.

- Select the *Portfolio* class and the *PORTFOLIO* table.
- Select the *customer* instance variable and the *CustomerId* column.
- Activate menu item *Variable* þ *Persistent 1→1 Relationship...*. In the dialog window which appears, the only foreign key relationship that appears in the main list is to the *CUSTOMER* table.
- Select this foreign key relationship; the *Customer* class appears on the right-hand side below Target Class / Variable. Select the class *Customer*, not the subclass XmpCustomer. After selection, a list appears in the lower part which contains all the instance variables still available for the target side of the

relationship. *Portfolio* should also appear here:

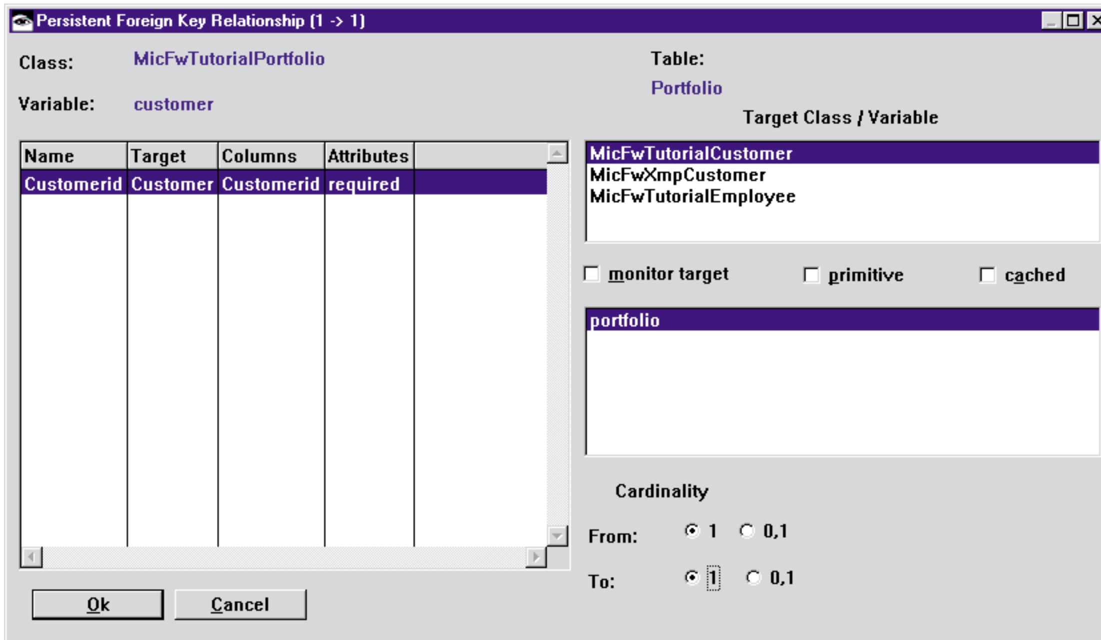


Figure 14: The 1:1 Relationship dialog window

- Select cardinalities *From:*1 and *To:*1.
- Select *portfolio* and confirm your choice by clicking on *Set Relationship*. As a check, select the *Customer* class and check to see if a 1:1 relationship to *Portfolio* from instance variable *portfolio* has now been implicitly created.
- Next, proceed with the 1:N relationship between an employee and a customer. As a convention, this will always be defined as being viewed from the :N side of the relationship.
- Start with the *Customer* class and the *CUSTOMER* table.
- Select instance variable *ownerEmployee* and column *EmployeeId*.
- Activate menu item *Variable* þ *Persistent N→1 Relationship...*. In the dialog window which is displayed, the only foreign key relationship that appears in the main list is to the *EMPLOYEE* table.
- Select this relationship; on the right side, *Employee* appears as a target class and, depending on the selection, a list of target instance variables appears.
- Select *ownedCustomers* and confirm your choice. Check the two-sidedness of the relationship as described above.
- Continue with the mapping of the N:1 relationship between the classes *XmpCustomer* and *XmpOrder*.
- Select the class *Order* and the table *XmpORDER*
- Select the instance variable *customer* and the column *XmpCustomerId*
- Activate menu item *Variable* þ *Persistent N→1 Relationship...*. Select the foreign key to table CUSTOMER
- In Target Class /Variable select the class *XmpCustomer* and the variable *orders*
- Select *Set Relationship*. Again check the two-sidedness of the relationship.

Finally, we come to the N:M relationship between a person and his or her address. N:M relationships assume the existence of an association table that connects the two tables. In the example, the *RESIDENCE* table connects the *PERSON* and *ADDRESS* tables using the corresponding foreign keys.

In the case of an N:M relationship, you can start with either side. Proceed, for example, as follows:

- Select the *Person* class and the *PERSON* table.
- Select instance variable *addresses*.
- Activate menu item *Variable* þ *Persistent N→M Relationship...*. In the dialog window that appears, the only table displayed in the list on the left is *PERSON*. Select this table and then do the following in the order given: select *Residence* to be the *associative table*, *Personid* and *Addressid* as the *left* and *right foreign keys*, then *Address* as the *Target Class* and *persons* as the *Target Variable*. Confirm your selec-

tions by clicking on *Set Relationship*.

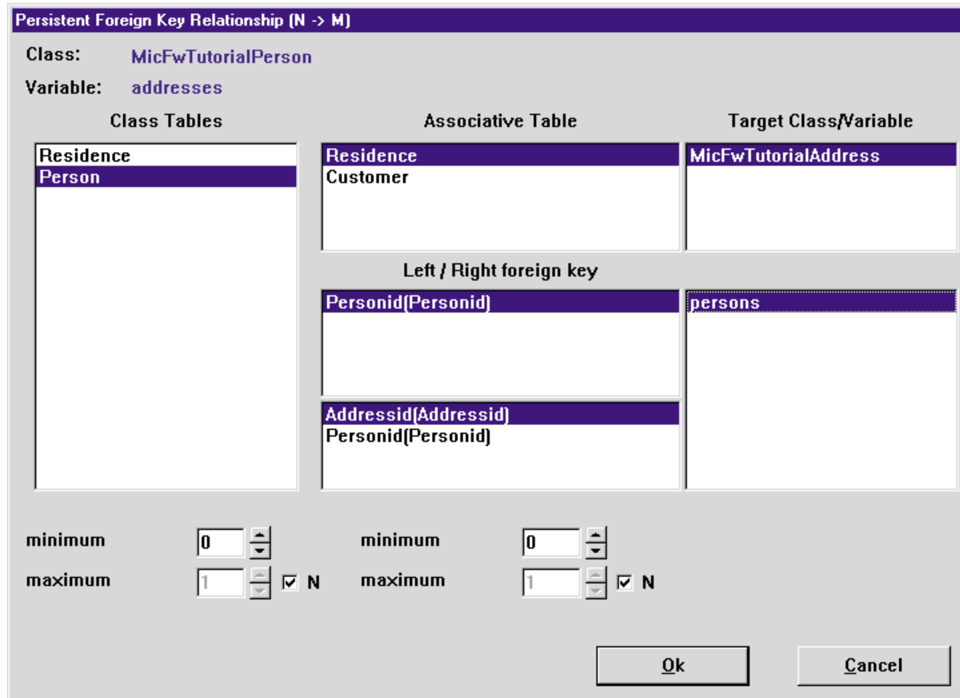


Figure 15: The N:M Relationship dialog window

Finally, you should store the POM (*Manager*þ *Store Globally*) and re-check all the mapping. The PFW helps you to validate your work using *Manager*þ *Validate Mappings...*. If everything is O.K., you should exit the STOPF dialog window and save the image.

2.3.8. Aggregation relationship

The name of a person is modeled in a separate class *PersonName*. In this example PersonNames shouldn't be stored in a separate table. The instance variables firstName and lastName of class Person-Name will be stored in the table PERSON See 'Aggregated 1-to-1 relationship' (page 81).

- Select the class Person
- Select the instance variable *name*
- Use *Variable Aggregated 1→1 Relationship*. The relationship editor opens.
- Select the class *MicFwTutorialPersonName* as target class of the relationship and select Primitive as the relationship type.
- Press the Continue... button. The Aggregation Mapper opens as shown in the following figure.

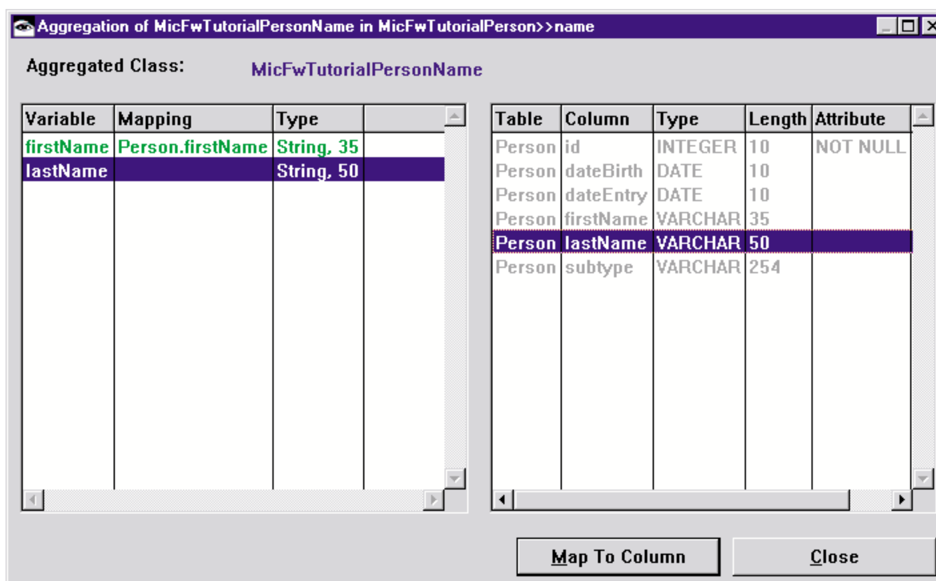


Figure 16: The Aggregation Relationship dialog window

- From the left list select instance variable *firstName* of class *PersonName*, from the right list select column *firstName* of table PERSON and click *Map To Column*. Repeat this step for the instance variable *lastName* and column *lastName.*
- Select *Close*

2.3.9. User-defined relationships

Standard behavior is not always required after the PFW has resolved relationships. Classes ***MicFwXmpOrderLine***, ***MicFwXmpProduct*** and ***MicFwXmpProductGroup*** provide an example of this. A product whose definition changes may not simply be overwritten, but must be versioned, since every OrderLine must always know the precise state of the current product at the point in time when this OrderLine was created. In other words, a product is never deleted, but versioned. All the versions of a product represent its history. If a ProductGroup receives an inquiry about its products, it only returns the current version of each product. This type of relationship must be defined as a User-Defined Relationship in STOPF.

Make sure that the primaryKey of the *PRODUCT* table comprises the *id* and the *historyId*. The history of a product thus comprises all entries with the same id; the latest version can be identified by the highest historyId. In the example, you can see that **no** FK relationships are defined in the database schema for UDRs. This means that the required object relationship must be created using an additional attribute *idOfProductGroup* in the Product class .

2.3.9.1. Creating the Mapping

In order to create this mapping, proceed as follows:

- Select class ***MicFwXmpProduct*** and variable *productGroup* in STOPF. Open the relationship editor using *Variable* þ *User-defined Relationship* Open the *User-defined Relationship Specification* window by clicking on *Continue....*

2.3.9.2. Definition of left specification (MicFwXmpProductGroup >> products)

By using *New...* and selecting *Static Sql Clause*, open the *Static SQL Clause specification* window on the left-hand side (left specification). This lets you define the WHERE clause with which the ProductGroup (products) → Product relationship must be resolved (this example is restricted to the use of **StaticSql Clauses**; for the **Dynamic** option, see ‘Dynamically defined user-defined relationships’ (page 82).

- First of all, two database alias names must be defined. In order to create the first alias, enter **x** in the Alias field, select the Product entry from the Table drop-down list and press the *Add* button.
- In order to define the second alias, press *New*, enter **y**, select @ from the Table list and press *Add* (the second alias is initially linked using the placeholder symbol @ and is only allocated later, in the SubSelect).
- The WHERE clause must be defined next. In the 'WHERE clause' window area enter:
  x.idOfProductGroup = ...
- Then, using *Add Variable,* add the *id* variable. This supplies the id value of the relevant instance of ProductGroup at run time.
- You then get the following:
  x.idOfProductGroup = ? id ...
- In order to specify that only the product with the highest historyId is relevant, proceed with:
  ... **and x.historyId = (select max(historyId) from XmpProduct y where x.id = y.id)**

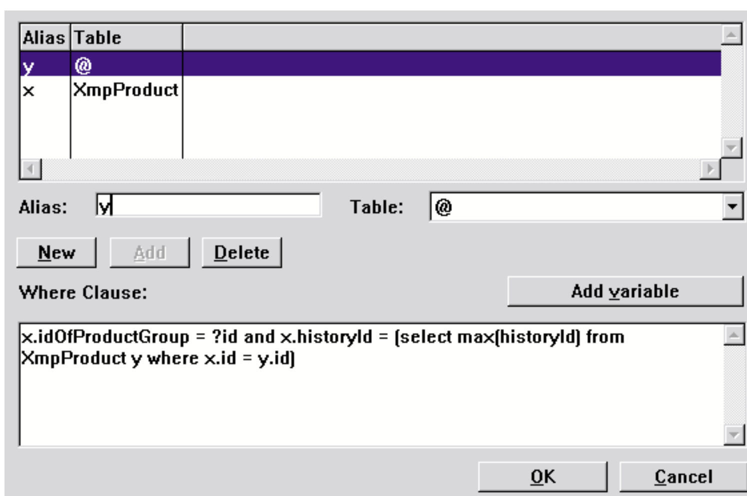


Figure 17: UDR Static SQL Clause left specification

Confirm with OK.

2.3.9.3. Definition of right specification (MicFwXmpProduct >> productGroup)

- Open the *Static SQL Clause specification* window on the right-hand side (right specification) by using New... and selecting *Static Sql Clause*. This lets you update the SQL statement with which the Product(productGroup) -> ProductGroup must be resolved.
- Only one alias is required here. Create alias x for the ProductGroup table.
- Define the WHERE clause as follows:
- Enter**x.id =**
- and then add the *idOfProductGroup* variable using *Add Variable*. The following text will appear: **x.id = ?idOfProductGroup**

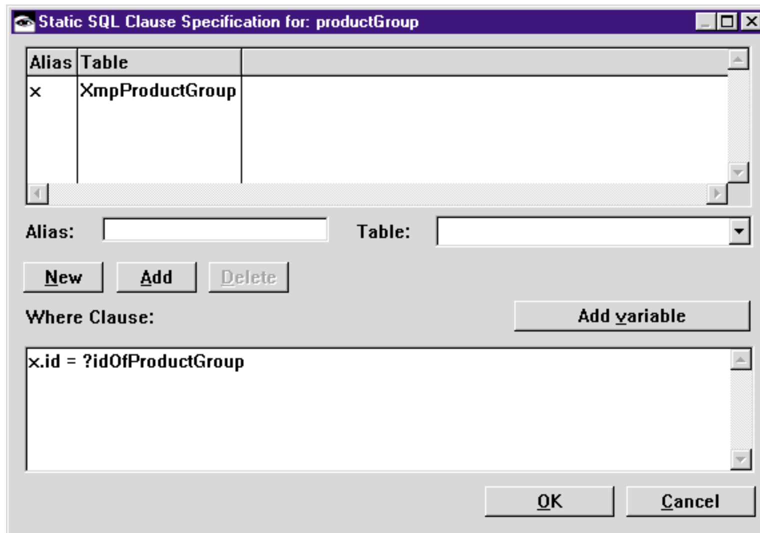


Figure 18: UDR Static SQL Clause right specification

- Confirm with OK. The display then looks as follows:

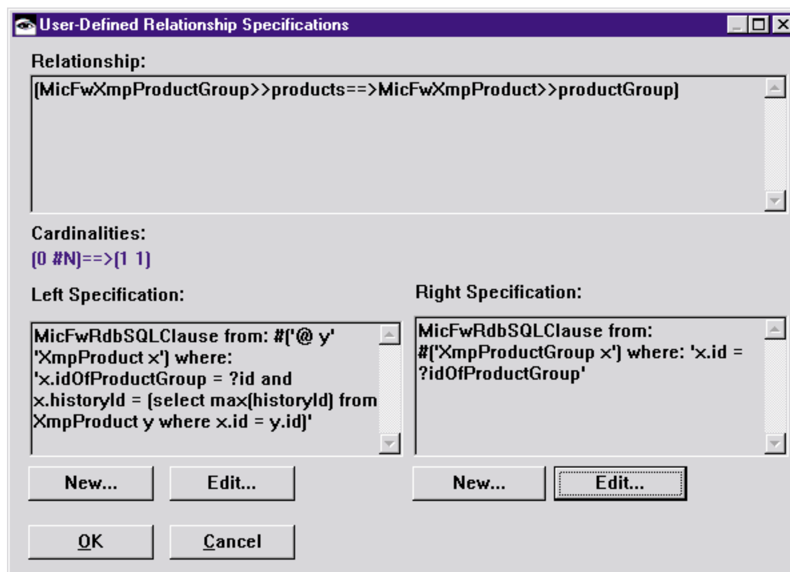


Figure 19: UDR Specification Dialog

2.3.9.4. Specialization of the persistenceInsert: and persistentUpdate: method in MicFwXmpProduct

Since no FK relationships are mapped for UDRs, it is up to the application to map this relationship. In order to correctly set the value of the *idOfProductGroup* attribute, the methods *persistenceInsert:* and *persistenceUpdate:* must be specialized in the Product class.

```
persistenceInsert: aVersion
"Private - copy the redundant relationship attributes from <aProductGroup>"
| aProductGroup |
(aProductGroup := productGroup getLoaded) notNil ifTrue: [
self idOfProductGroup: aProductGroup id.
```

```
].
persistenceUpdate: aVersion
"Private - copy the redundant relationship attributes from <aProductGroup>"
(aVersion hasVersionFor: #productGroup) ifTrue: [
aVersion recordNewVersion: self productGroup id
for: #idOfProductGroup
].
```

2.3.9.5. Code Examples

You will find examples of UDR usage in the Framework Transcript menu *micFrameworks* þ *Browse Examples* under ***MicFwPersistenceOdbcExamples*** in the *xmpNonStandardRelationship1* and *2* examples.

2.3.9.6. Summary: Differences to standard relationships

- In the class model, an attribute which simulates this relationship at object level (idOfProductGroup in the example) must be created in the class in whose table the FK of the User-Defined-Relationship would be defined in normal (standard) relationships.
- When using an UDR, the application must instantiate the *persistenceInsert:* and *persistenceUpdate:* methods in order to ensure that the object relationship concerned is updated before being written to the database.

2.3.10. Optimistic concurrency

In contrast to pessimistic data locking, Optimistic Concurrency enables maximum data availability. The updated data is not locked to other users, which means that conflict situations may occur. In order to detect these conflict situations you may use *timestamp columns*. A detailed technical description of this feature can be found in 'Optimistic Concurrency' (page 151).

In this chapter, we will show how the different Optimistic Concurrency strategies can be used, and how conflict situations can be resolved. This will be explained using the example of classes ***MicFwXmpProduct*** and ***MicFwXmpProductGroup***. In order to do this, generate a Persistent Manager just for these two classes:

- select class ***MicFwXmpProduct*** in the POM generator,
- select option 'with all related classes',
- enter 'MicFw' into the Ignore prefix entry field,
- then set the "abstract (suppress table)" attribute for class ***MicFwTutorialIdentifiedObject***.
- generate the POM.

Different Optimistic Concurrency strategies will now be applied for the two example classes in STOPF.

2.3.10.1. Set Optimistic Strategy for MicFwXmpProduct

The mapping for the MicFwXmpProduct class should be set such that the comparisons required for optimistic concurrency take place with a timestamp specification. A (mapped) column in the Product table must be given an **optimistic strategy** in order to do this.

- Select the timestamp attribute mapped to the c_timestamp column and double-click on it. The Attribute Editor opens.
- Select "*Value Message*" with selector *timestamp* as the INSERT and UPDATE value strategy, which fetches the new timestamp from the database once only (when starting a commit for a new context).
- Select "Check always" as the optimistic strategy - so the timestamp column always appears in the WHERE-clause of SQL UPDATE statements generated by the Framework.

A Reload Strategy is not necessary here, since the new timestamp values are known when the update/insert takes place, making another database access unnecessary. The following figure shows the result

after setting the attributes.:

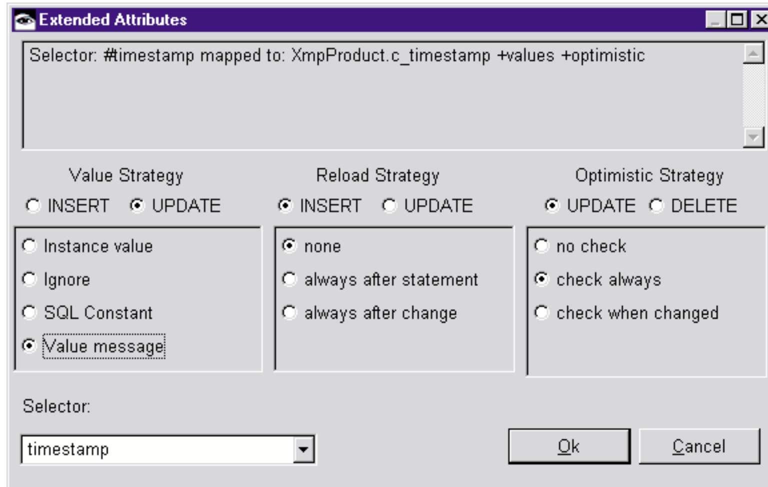


Figure 20: Attribute settings for MicFwXmpProduct>>timestamp

2.3.10.2. Chapter Set Optimistic Strategy for MicFwXmpProductGroup

- Select class ***MicFwXmpProductGroup*** and double-click on the instance variable ***#timestamp***. The attribute editor opens.
- Select "*SQL Constants*" with constant 'CURRENT TIMESTAMP' as the INSERT and UPDATE value strategy.
- Select "Check always" as the optimistic strategy.
- Select "Always after statement" as the INSERT and UPDATE reload strategy.

The following figure shows the result after setting the attributes.

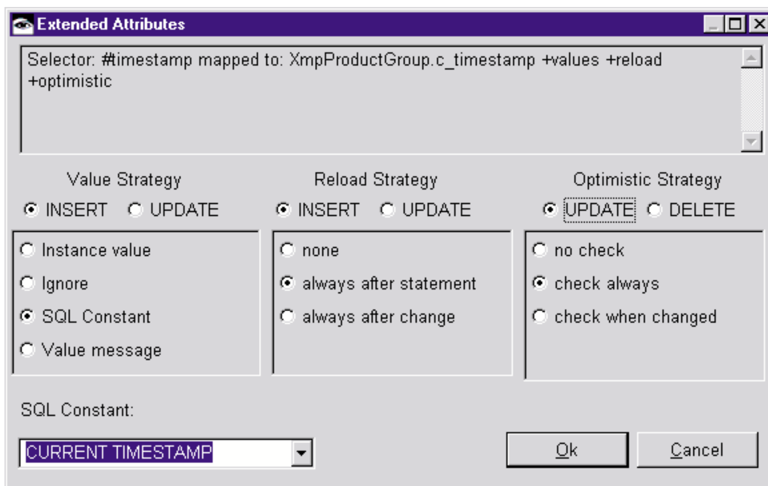


Figure 21: Attribute settings for MicFwXmpProductgroup>>timestamp

2.3.10.3. Configure POM

In order to activate the Optimistic Concurrency setting, the POM must be configured accordingly. See also ‘Choosing the POM's commit handler’ (page 160).

- In order to do this, open the configuration editor in STOPF using *Manager Configuration*.
- In the 'Commit Handler' section, select ***MicFwRdbCmtHdlrOptimistic*** and apply.

In section ‘Option Compare’ select ‘ON’ and apply.

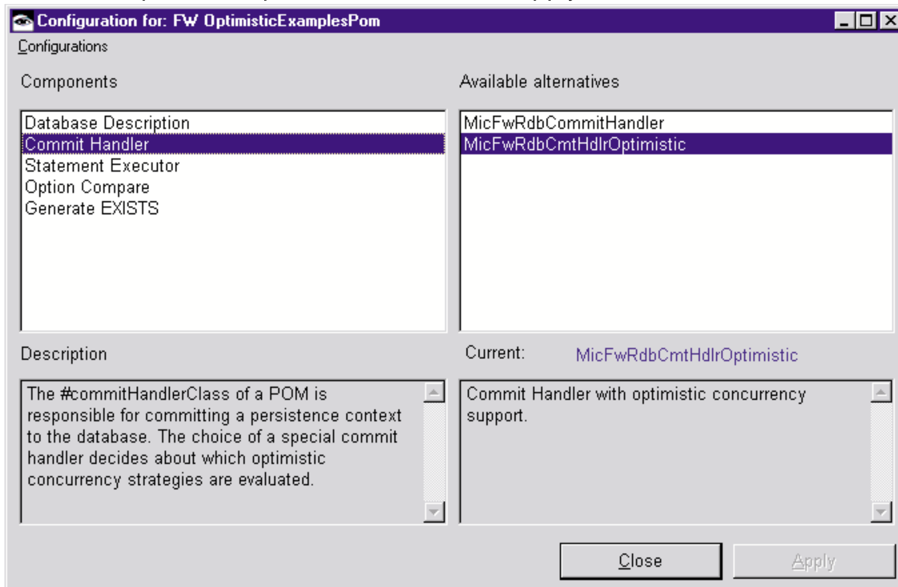


Figure 22: POM Configuration

There is still one preparatory step to perform:

- The new timestamp value is fetched from the database before each write access for class ***MicFwXmp-Product*** to the database by sending the #timestamp message to the POM's commit handler. This occurs using a predefined SQL statement which the POM must know about. This prefetch statement must be entered in the POM's registry under the name #getServerTimestamp (see also ‘Configuring classes for optimistic concurrency’ (page 156) an appropriate value message). In order to do this,
- use the POM’s Statement Registration *(Manager Statement Registration...)*.

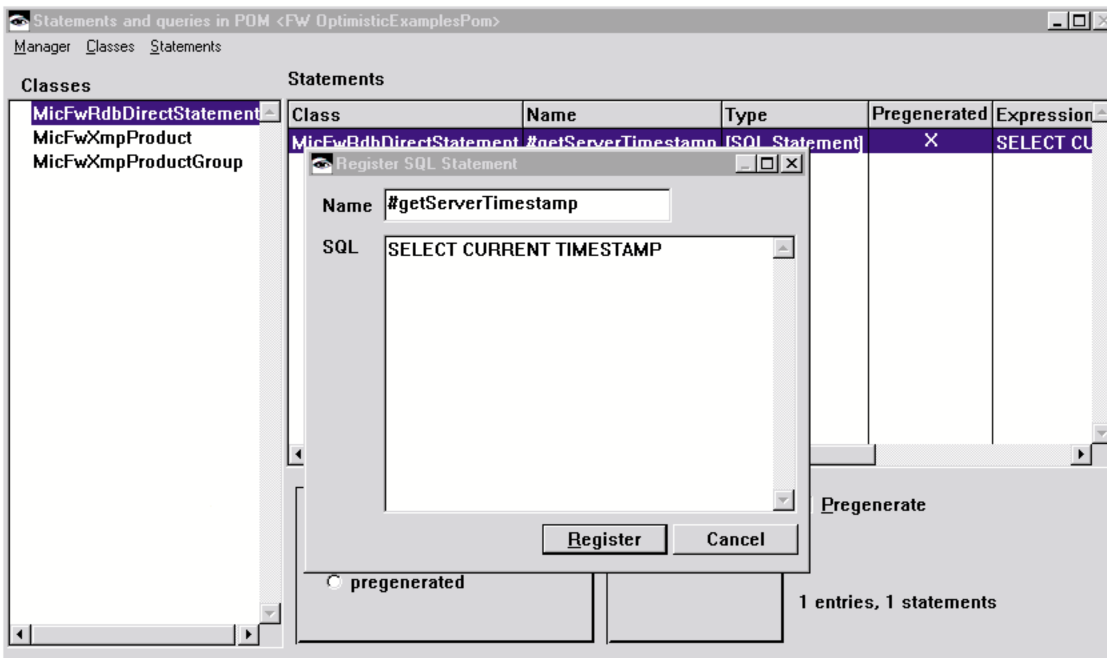


Figure 23: QueryRegistration #getServerTimestamp

- Select the entry ‘***MicFwRdbDirectStatement***’ and click ‘Statements -> New...’ to register a new statement.
- In the ‘Name’ field enter **#getServerTimestamp**
- In the ‘SQL’ field enter**: SELECT CURRENT TIMESTAMP.**

Note: Under DB2, the statement can be **'SELECT CURRENT TIMESTAMP FROM SYSCAT.TABLES'** or **'VALUES (CURRENT TIMESTAMP)'.**

- Store the POM using Manager Store Globally As ....
- Then export the database schema and create the database for this example.

(You can also use the POM named ***MicFwExamplesPomOptimist*** selecting *micFrameworks Browse Per-*

sistence Manager).

In order to be able to see the effect of the selected settings, modify an existing instance of the ***MicFwXmpProductGroup*** class. Execute the following example in a workspace. Then look at the generated SQL statements in the Framework logger.

```
Smalltalk
   at: #ThePOM
   put: (MicFwPersistenceManagerOdbc named: <name Of your pom>).

| aContext aProductGroup |

ThePOM isConnected
   ifFalse: [ThePOM connectWithDriverPrompt].

"begin a new transaction ... "
aContext := ThePOM newPersistenceContext.
aContext beginTransaction.

"create a new ProductGroup and store it in the database ..."
aProductGroup := MicFwXmpProductGroup newPersistent
   id: 100;
   name: 'Old Name';
   yourself.

"commit the first transaction"
aContext commitTransaction.

"start a second transaction ... ."
aContext beginTransaction.

"load the new productGroup from the database, change its name and commit again
... "
(ThePOM persistenceIdentifierFrom: aProductGroup) get name: 'New Name'.

"commit the transaction"
aContext commitTransaction.
```

The following entries have now been made in the logger:

In the first context:

- an INSERT statement with the CURRENT TIMESTAMP expression for creating the new instance.
- a SELECT statement to get the new timestamp from the database (because of the reload strategy for ProductGroup>>timestamp)
- In the second context:
- the UPDATE statement with the CURRENT TIMESTAMP expression, in order to make the changes persistent
- another SELECT statement to get the current timestamp value from the database

Now we need to provoke a conflict. We can do this by updating an instance in a persistence context and by modifying this same instance using the SQL Direct Interface before the context is committed.

```
Smalltalk
   at: #ThePOM
   put: (MicFwPersistenceManagerOdbc named: <name of your POM>.

| aContext aProductGroup anIdentifier|

ThePOM isConnected ifFalse: [ ThePOM connectWithDriverPrompt ].

"begin a new transaction..."
Context := ThePOM newPersistenceContext.
aContext beginTransaction.

"load an existing ProductGroup ... "
anIdentifier := ThePOM persistenceIdentifierFor: MicFwXmpProductGroup.
```

```
anIdentifier setValueFor: #id to: 100.
aProductGroup := anIdentifier get.

"Now change this ProductGroup using the Direct SQL Interface, to simulate
another user saving his modifications made to that ProductGroup ..... "
ThePOM executeSQL: (
'UPDATE XmpProductGroup SET c_Timestamp = NULL WHERE id = %1' bindWithArgu-
ments: (Array with: aProductGroup id asString)
).
ThePOM commitDatabaseTransaction.

"change the productGroup .. "
aProductGroup name: 'Another Name'.

"commit your work..."
aContext commitTransaction.
```

A ***MicFwPsUpdateError*** is signaled, because something belonging to the Timestamp information has been changed on the database (the instance has been modified or deleted).

Examples of a code for handling update errors can be found in the Framework Examples Browser (under *MicFwPersistenceOptimisticConcurrencyExamples*).

2.4. OQL

In order to see how simple OQLs are dynamically and statically created, you can create and analyze OQLs yourself using the persistent tutorial application. The persistent tutorial is started by entering the following command:

```
MicPsTutorialPersonListEditor open
```

Note: The tutorial works with a POM that has been stored under the name of ***FW Examples***. If your POM has been stored under another name, you can use the STOPF tool to globally save this POM under another name.

After starting the tutorial, the following window appears:

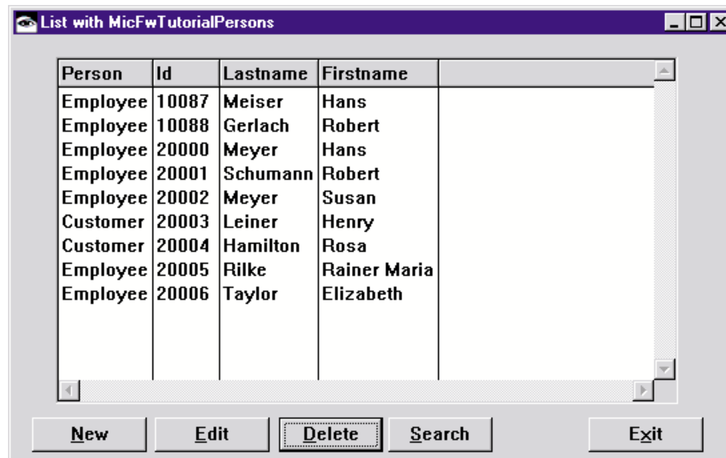


Figure 24: Persistent-Tutorial MicPsTutorialPersonListEditor

In this list editor you can create, modify and delete new Persons. Clicking on the *Search* button puts you into the OQL browser, in which you can create OQLs. Before the browser opens you must select another class from which you would like to create a query (e.g. **MicFwTutorialPerson**). Then the OQL Browser appears:

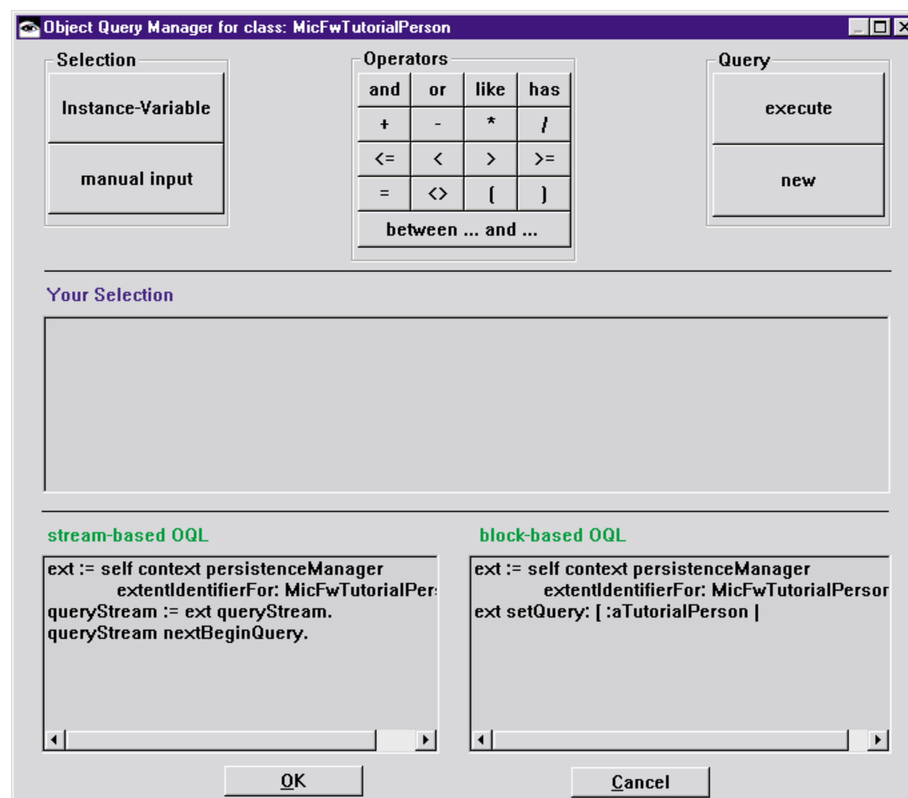


Figure 25: OQL-Browser

The instance variables for the associated class are displayed by clicking on the *Instance Variable* button. If you select a relationship, the associated instance variables are displayed immediately.

For example, values for comparisons are entered using the *manual input* button.

In order to run through an example, perform the following steps:

- click on ***Instance Variable***
- select ***name*** and then ***lastName***
- click on the ***like*** **button:**
- click on the ***manual input*** button and enter ***M%***
- the query is completed and executed by clicking on ***execute***. After execution, an inspector containing the result of the query is opened

The OQL browser then looks as follows:

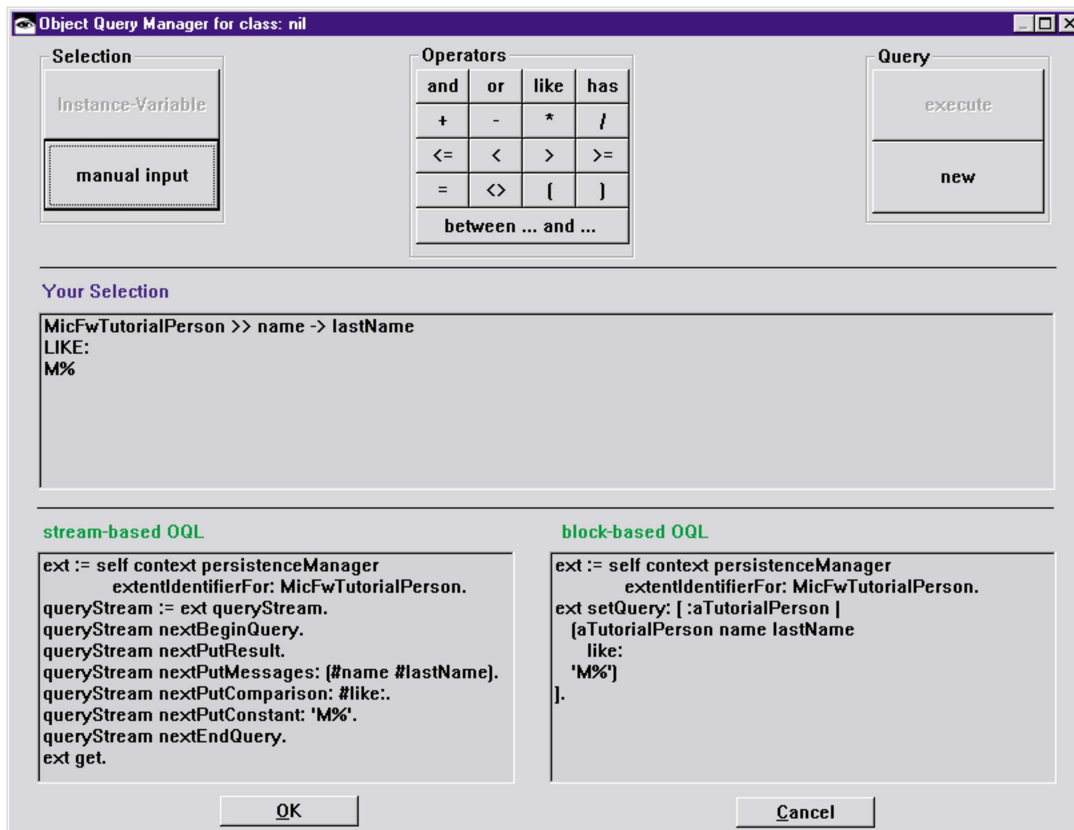


Figure 26: OQL-Browser after execute a sample query

The lower area contains two windows that display the current OQL in two different forms:

- The left-hand window displays the **stream-based OQL**. This is very useful for dynamic OQLs, i.e. the OQL is not created and analyzed until run-time. The OQL browser also uses the stream-based OQL in this tutorial.
- The right-hand window displays the **block-based OQL**. This type of OQL is used for statistical queries, i.e. the query is known about at run-time, with the possible exception of the query value such as the ***'M%'*** in ***lastName*** in this case (see ‘Query Parameters’ (page 137)).

The browser is re-initialized by clicking on ***new*** and a new query can be set up.

Note: The tutorial is split into two applications: The first is the **MicObjectFrameworkTutorial**, in which the tutorial object behavior is described, and the second application is the **MicPersistenceFrameworkTutorial**, which contains the Persistence Framework.

3

Mapping Guide and Reference

3.1. Introduction

In reasonably complex models there will often be a number of alternative paths to take when mapping objects to relations. These range from simply generating one model from another to hand-mapping the different structures in order to meet special project requirements.

This chapter contains detailed conceptual descriptions of the individual items that together form the mapping of an object model to a relational model.

A good understanding of the concepts described will enable the designer to create optimal mapping between the two worlds and thus affect several aspects of the resulting application, such as

- application structure
- application performance
- The steps for creating the mapping are described in the documentation for the tools provided for that task.

3.2. Instance Variable to Column

Instance Variable to Column Mapping (Instance Variable Mapping) is the most basic form of mapping.

It defines a direct correspondence between a given instance variable and a column in a table.

Thus the value found in the instance variable is stored directly in this column whenever changes from the object model are propagated to the DBMS.

Since the database column can only contain values of primitive types (e.g. String, Integer), the instance variable must either contain an appropriate value or a conversion rule must be defined for the value in the instance variable that defines a conversion to the target type.

The features of the Object Behavior Framework are used for typing and type conversion. For instructions on how to define conversions, see the appropriate chapter in [OBFW].

3.3. Subclassing

Subclassing is an indispensable practice in object oriented design and programming. However, it is one of the features where the relational model falls short. The PFW provides therefore explicit support for mapping class hierarchies to relational tables.

In general, a Smalltalk class hierarchy can be mapped to tables in the following ways:

- each class is mapped to an individual table. We use the term *Canonical Subclass Mapping*
- two or more classes are mapped to the same table. We use the term *Denormalized Subclass Mapping*
- Both alternatives can be mixed within the same hierarchy.

Two important issues that are also discussed in this context are *table connections* and *type discriminators*. For the following discussion, we will refer to three classes - ClassA, ClassB and ClassC - where ClassB is a subclass of ClassA and ClassC is a subclass of ClassB.

3.3.1. Canonical subclass mapping

We use the informal term *Canonical Subclass Mapping* if a class hierarchy is mapped such that every class involved only introduces new mappings to new tables (not inherited ones).

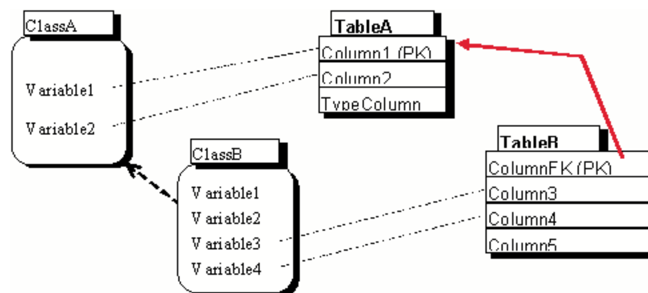


Figure 27: Canonically Mapped Class Hierarchy

ClassA is simply mapped to TableA. The primary key of TableA appears as an identifying foreign key in TableB. ClassB maps its non-inherited attributes to columns in TableB. ClassB also inherits the mappings that were created for ClassA.

An internal table connection (see chapter Classes Spanning Several Tables Classes Spanning Several Tables) is created to connect the corresponding rows from TableA and TableB. *TableA.TypeColumn* is declared as *type discriminator* (see chapter Type Discriminator Type Discriminator).

3.3.2. Denormalized subclass mapping

Denormalized Subclass Mapping describes mappings where two or more classes from a class hierarchy are mapped to the same table.

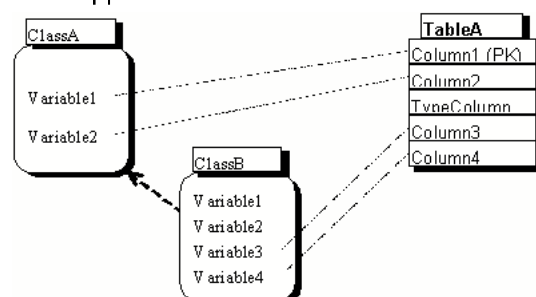


Figure 28: Denormalized Mapping of Class Hierarchy

This mapping requires no foreign keys. Retrieval performance can therefore be significantly better than for canonically mapped classes. The type discriminator column is still used.

Note: If an instance of ClassA is stored, the columns that are only mapped in ClassB are not written. They contain NULL values. A redundancy has thus been introduced into the data model.

3.3.3. Unified subtable load

There are two alternative load algorithms available for subtype classes, that can be defined for each class mapping.

You have the following options:

3.3.3.1. Tablewise subtable load (the default, as usual)

Normally when polymorphic load operations are executed, the generated statement first accesses the table(s) of the topmost class only. Example:

Class (mapped to) TABLE:

```
Person - PERSON
   Customer - CUSTOMER
   Employee - EMPLOYEE
pom loadAllInstancesOf: Person
```

This results in a `SELECT ... FROM PERSON`. The resulting cursor is traversed, and for every row the type of instance is determined based on the subtype value in PERSON. Then, the appropriate subtable is accessed with a single SELECT. Thus, for 200 Customers and 200 Employees (no Persons), the number of SQL statements executed will be 1+200+200. This can get extremely inefficient with large answer sets and deep multi-table inheritance trees.

3.3.3.2. Load with a UNION Statement (=Unified subtable load)

An alternative is to create a UNION of the load statements for all possible classes. This results in only ONE statement being issued to the DBMS and only one cursor being traversed.

Features

This feature currently is available for the following accesses:

- #loadAllInstancesOf:
- #getAllInstancesOf:
- Loading of to-1 and to-many relationships
- #persistenceIdentifierFor: with #load and #get

When loading an object with a persistenceIdentifier or when resolving a to-1 relationship, this new feature normaly has no effect, even if it has been activated. This is because, it normally does not improve performance when there are less than 4 SQL statements to be executed to load the object. Only if the average number of statements to load the object are more than 4 (e.g. for deep subclass hierarchies), the FW uses the UNION select statement, if it has been turned on for this class.

Subtable load and unified load can be used together in the same POM, so there should be no migration or backward compatibility problems.

To turn on "Unified subtable load", use the STOPF-tool menu item *Class→Subtyping→Unified subtable load*. If it has been turned on, load operations for this class use the UNION statement. Load operations for subclasses are not affected.

When storing a POM in a class or export file, a new method #option<Class>MappingIn: will be generated for each class mapping. The default for #optionUnifiedLoad is <false>.

Specifics of SQL generation with Queries

There is currently no effect for SQL generated from OQL queries

Remarks

to-1 load operations (N1 relationship or persistenceIdentifier) normally are faster with the usual tablewise subtable load algorithm. This could be different for deep hierarchies.

to-many load operations (#loadAllInstancesOf:, OQL, to-many-relationships) are faster with unified load, when a special minimum amount of objects will be loaded.

The break even point can be described as follows:

When you load two objects (two MicFwTutorialPersons: 1 MicFwTutorialEmployee and 1 MicFwTutorial-Customer) by resolving the to-many relationship of a MicFwTutorialAddress>>#persons, the unified load is twice as slow as the tablewise subtable load. This is because:

- The first SQL statement when using tablewise subtable is always very simple
- To load two objects, two additional subtable load statements are performed, so that there are 3 statements altogether
- The UNION select when using unified load is more complex, the more subclasses are in the hierarchy that have to be unified. The example POM for this test had five subclasses, what made the UNION statement slower than 3 simple SQL statements.

Conclusion: The break-even point for this example is for about 5 objects loaded from the relationship. This can differ for a different number of hierarchy levels.

3.3.4. Subclass Table Connections

The mapping for a given class comprises all mapping inherited from superclasses and mapping which is

done for the class itself. Therefore, if a class hierarchy is mapped over several tables, some classes in the hierarchy may be stored over several tables.

A connection path interconnecting all tables must be given for all tables involved in loading or storing an instance of such a class. This connection path is called an Internal Table Connection.

The same concept applies to classes that are mapped over several tables without inheritance. A separate chapter has therefore been created to cover this topic in detail (see chapter Classes Spanning Several Tables Classes Spanning Several Tables).

3.3.5. Type Discriminator

In the Smalltalk environment, inheritance and polymorphism are standard features of the language and are used extensively in most applications.

Relational database systems do not know about classes, and therefore do not support inheritance or polymorphism.

Looking at the example classes, when instances of ClassA are loaded from the DBMS in polymorphic mode (i.e. instances of ClassA *and* any subclass), there must be a way to distinguish rows from TableA that were written from an instance of ClassA and rows from TableA that were written from an instance of ClassB. This is achieved by using a column in TableA (called "TypeColumn" in our case) to store an indicator value for the class identity.

This is an entity relationship modeling practice known as using a *type discriminator column.* We will also use the term *subtype column*. The type of the object to be created after retrieval can be determined using the value in this column.

A type discriminator column must be specified for each class that has subclasses. The column is defined (mapped) at superclass level. For each subclass, a *type value* must be defined at the level of the respective class.

Note: A subtype column is not needed if polymorphic loading is not required.

Referring to our above example, column *TableA.TypeColumn* is mapped as a subtype column in ClassA. For ClassB, the type value is defined as 'B'. The PFW now knows that every row in TableA whose column TypeColumn contains the value 'B' belongs to an instance of ClassB.

A subtype column can be defined at each inheritance level or it can be used over several levels.

3.3.5.1. Restriction

If a subtype column is used over several levels, the PFW currently has a restriction that should be taken note of. For each class in such a hierarchy that has a type value assigned, referencing that class in a polymorphic operation will only affect instances of that class alone, not subclasses. For example, consider the following class hierarchy:

Class	Mapping
Person (abstract class)	subtype discriminator column TA.CTYPE
Customer (concrete subclass of Person)	type value: 'CUST' defined for TA.CTYPE
InternalCustomer (concrete subclass of Customer)	type value: 'ICUST' defined for TA.CTYPE

Table 6: Example for subtyping

Now let the Employee class have a persistent relationship to Customer (Employee>>customers<->Customer>>employee). Let an instance of Employee and InternalCustomer be created and the relationship be set between the two. If the employee is now loaded into the image at a later time, and the #customers relationship is traversed, the PFW attempts to load the associated Customers. Since Customer is a concrete class, the type value for that class will be used by the PFW for the retrieval operation. Since the actual Customer is an InternalCustomer, it was stored with a type value 'ICUST'. It will therefore not be found.

Workaround: assign a separate subtype column for each concrete class with subclasses.

3.4. Classes Spanning Several Tables

In the PFW, classes can be mapped spanning several tables. This can either occur explicitly for one class, or it can occur if a class inherits mappings from superclasses. In both cases, a connection path that connects all tables must exist. This connection path is then used to find the corresponding rows from all tables that together make up the data of an instance for the respective class.

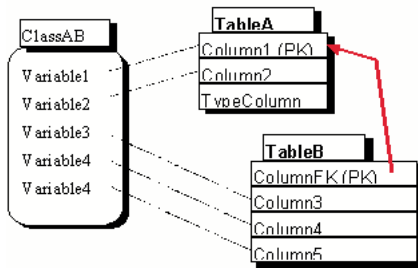


Figure 29: Single class spanning several tables

The connection between two tables is called an ***Internal Table Connection***. On the RDBMS side, a connection between two entities is implemented through a foreign key. The definition of an Internal Table Connection must therefore be based on a foreign key (note that foreign keys can be manually declared to the PFW). The foreign keys thus defined provide the "join condition" that enables the PFW to find corresponding rows.

Example:

SQL statement generated by the PFW to retrieve data for an instance of ClassAB with id value 22. The italicized text shows the join condition.

```
SELECT a.column1, a.column2, b.column3, bcolumn4, b.column5 FROM TableA a,
TableB b
WHERE a.column1 = 22 and a.column1 = b.columnFK
```

3.5. Foreign key based relationships

One of the extensions to the Smalltalk object model that the Mynd Frameworks have introduced is the explicit declaration and management of relationships between objects. This is an extension that becomes especially useful in connection with persistence.

As noted earlier, the relational model normally implements entity relationships using foreign keys. In order to make relationships persistent, there must therefore be mapping between foreign keys and object relationships. This chapter covers the details of mapping object relationships to foreign keys.

3 kinds of relationships are considered, based on cardinality:

- 1-to-1
- 1-to-Many
- Many-to Many

Each relationship is supported as a bedirectional (two-sided) or unidirectional (primitive, one-sided) relationship. 1-to-1 relationships can be represented by a foreign key or by aggregation mapping. Aggregation mapping is covered in chapter Aggregated 1-to-1 relationship Aggregated 1-to-1 relationship. 1-to-Many and Many-to Many relationships must be represented using foreign keys. An additional associative table is required for Many-to-Many relationships.

In terms of mapping, the differences between 1-to-1 and 1-to-Many relationships are limited to the persistence tool support and the object model. For the rest of this discussion they will therefore be treated as one.

Restriction: A mapping for a two-sided relationship from one class to the same using the the same attribute as a back reference is currently not supported.

3.5.1. Foreign key based 1-to-1 and 1-to-Many relationships

If a 1-to-1 or 1-to-Many relationship is mapped, it is associated with a foreign key in the underlying database model. This enables the PFW to retrieve the instances referenced by the relationship by generating appropriate SQL statements that include a join condition for the foreign key. For foreign-key based relationships it is useful to distinguish between both sides of the relationship in terms of dependency:

the class that is mapped to the table holding the foreign key is called the *child class*

the class that is mapped to the table referenced by the foreign key is called the *parent class*

Note: A class can be in parent role in one relationship and in child role in another.

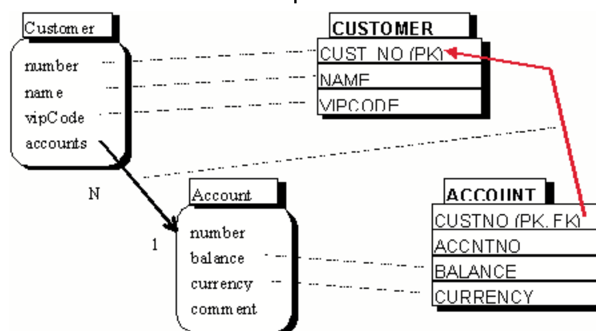


Figure 30: Foreign Key Based Relationship

3.5.1.1. Changing relationships

Depending on whether the relationship is loaded in the child→parent or parent→child direction, the appropriate SQL statements are generated. If the relationship is changed, the changes are propagated accordingly to the underlying columns. The following updates may apply:

- If a relationship is established with a previously stored child object, an update is generated for the foreign key columns
- If a relationship is established with a newly-created child object, the insert operation also contains the foreign key values
- If a relationship is removed, the foreign key columns of the child object are set to NULL

Note: When an instance of the child class is loaded into the image, the values of the foreign key columns underlying the relationship are not stored in instance variables. Instead they are stored with the persistent object's persistenceProperties. If an application requires access to these values (which should happen rarely), any persistent object can be sent ***persistenceProperties nativeObjectFor: thisInstVar***, which returns an array of values.

3.5.1.2. CASCADED delete

In most relational DBMS, foreign keys may have associated UPDATE and DELETE attributes. The attribute directly supported by the PFW is the *ON DELETE CASCADE* attribute.

If the parent object of a relationship whose underlying foreign key has this attribute is deleted, the delete operation is cascaded to all child objects that are currently loaded by also marking them as deleted. However, the delete operation for the child objects is not executed against the DBMS. Instead, the persistent data for the child objects is expected to be deleted implicitly by the DBMS.

Note: The PFW does not recognize cascaded relationship updates.

3.5.2. Many-to-Many relationships

In a relational database model, a Many-to-Many relationship is implemented by an associative table that holds foreign keys referencing both participating entities.

In the PFW, Many-to-Many relationships are therefore mapped to associative tables. If a relationship mapped to an associative table is traversed, the appropriate join condition across the three participating tables is generated and executed.

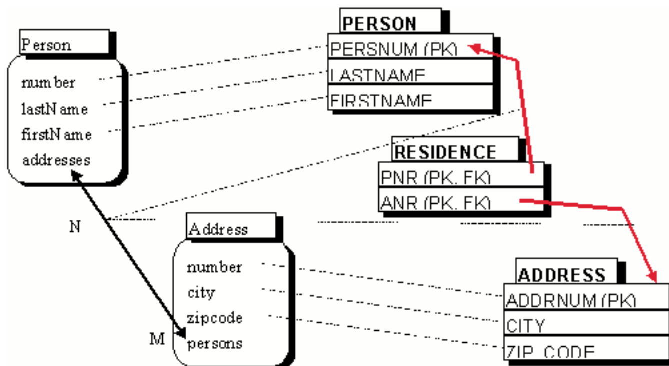


Figure 31: Many-to-Many Relationship

3.5.2.1. Changing N-to-M relationships

The following changes apply to Many-to-Many relationships:

- If a new relationship is established, a row is inserted into the associative table
- If a relationship is removed, the row is deleted from the associative table

3.6. Aggregated 1-to-1 relationship

There are cases where a to-1 relationship which exists in the object model is not complemented by a foreign key relationship in the database schema. This can occur either because the object model is mapped to an existing DBMS model, or because the decision has been made that implementing the relationship using a foreign key would incur too much overhead for DBMS access. In that case it may be feasible to combine both entities in one table, thus performing another type of denormalization.

Note: to-Many relationships cannot be implemented other than by using a foreign key.

The PFW allows the mapping of uni- or bidirectional to-1 relationships in such a way that both the source object and the target object are stored in the same table. This is called an *Aggregated Relationship Mapping*. Aggregated relationships do not require foreign keys to exist.

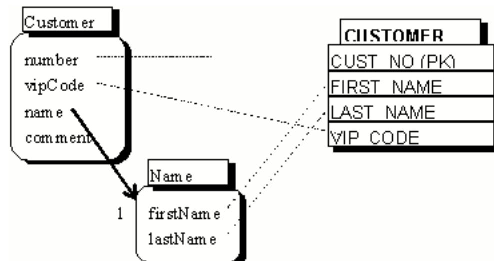


Figure 32: Aggregated Relationship

The main differences between these and foreign key based relationships are:

- no foreign key required

Note: A class that is mapped as the target of aggregated relationship mapping is subject to certain restrictions within the context of the associated persistence manager:

- Instances of the class cannot be made persistent by themselves. They are made persistent implicitly by assigning them to the enclosing object. Therefore an instance of an aggregated class may not be sent *newPersistent* or *becomePersistent*.
- Instances of the class cannot be retrieved individually from the DBMS
- A class once mapped as an aggregate can have additional aggregate mappings within the same POM, however it cannot also be fully mapped to a table
- The aggregated class cannot have persistent relationships originating from it.
- Use "monitorTarget", otherwise you loose UPDATES.

Note: Assigning the same (identical) instance of an aggregated class to different enclosing objects may lead to errors when using the DB refresh functionality. Situations like the following should therefore be avoided.

Example

```
aggregate := MyAggregateNameClass new.
person1 := MyPerson newPersistent
person2 := MyPerson newPersistent.
person1 name: aggregate."assign the same name to both persons"
person2 name: aggregate.
```

The following code is safer:

```
aggregate1 := MyAggregateNameClass new.
aggregate2 := MyAggregateNameClass new.
person1 := MyPerson newPersistent
person2 := MyPerson newPersistent.
person1 name: aggregate1."assign individual names to both persons"
person2 name: aggregate2.
```

3.7. User-defined relationships

Abbreviations	
FKR	Foreign Key Relationship
UDR	User-Defined Relationship

Table 7: Abbreviations for the chapter User-defined relationships

3.7.1. What are they?

User-defined relationships provide a means of implement persistent relationships within the PFW that are not based upon foreign keys. Instead, user-defined relationships are resolved by means of arbitrary user-defined SQL statements. It is thus possible to define relationships between entities that cannot be described as join conditions between tables.

For example, this may be necessary if the interpretation of column values depends upon other values that are only known at runtime.

Another example is the case where version timestamps are kept with the rows in some tables. In such cases, the timestamp will be part of the primary key. If a table contains a reference to another, however, the timestamp is not included. Instead, the reference implicitly points to the most current row. The foreign key definition is therefore incomplete (and not valid in relational terms).

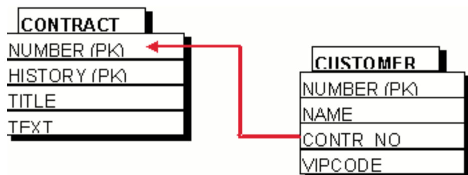


Figure 33: CONTRACT (versioned) and CUSTOMER

The CUSTOMER.CONTR_NO column is a "partial foreign key" to CONTRACT. The relationship Customer>>contract→Contract must be resolved by adding a (rather complex) subselect clause to the usual join condition:

```
SELECT a.number, a.history, a.title, a.text
FROM CONTRACT a
WHERE a.number = ?
and a.history = (select max(history) from CONTRACT c
where c.number  = a.number)
```

The "?" contains a value that has been retrieved from CUSTOMER.CONTR_NO.

This kind of statement cannot be generated automatically by the PFW. User-defined relationships therefore provide a means of defining relationships based on arbitrary SQL strings, which must be provided by the application developer.

User-defined relationships come in 2 flavors:

- dynamically defined UDR
- statically defined UDR

3.7.2. Dynamically defined user-defined relationships

In dynamically defined user-defined relationships, the SQL clause used to resolve the relationship is obtained by the PFW at runtime by sending a predefined message to the source object of the relationship.

For example, traversing the relationship Customer>>contract→Contract would result in a predefined message (e.g. *sqlClauseForContract*) being sent to the instance of Customer. This message must return an instance of ***MicFwRdbSQLClause*** which contains the full definition of the SQL WHERE clause or ***nil***. When this message returns nil, the framework assumes, that the relationship has no target objects assigned. No SQL statement will be executed when resolving the relationship.

Thus the SQL clause can be configured at runtime depending on values in the Customer instance.

You may define a message with one argument for the dynamic user defined relationship. The relationship instance itself will be passed as the parameter value to the method, which allows you to determine the according relationship in the callback method.

The message sent to the source object is defined at development time using the STOPF tool. See 'Dynamic specification editor' (page 202).

3.7.3. Statically defined user-defined relationships

For statically defined user-defined relationships, the SQL clause used to resolve the relationship is defined at development time using the STOPF tool. See 'Static SQL clause specification editor' (page 202). It is stored in the POM.

At runtime, upon traversing the relationship the PFW will construct the SQL statement from the base mappings of the associated classes and the predefined relationship SQL clause.

In contrast to dynamically defined user-defined relationships, the SQL for static user-defined relationships can be pregenerated and will be cached in the POM whenever generated. They can therefore also be executed in the embedded SQL mode provided by the PFW.

3.7.4. Defining relationship SQL clauses

The SQL clauses used to resolve user-defined relationships must be defined in a special way. This is due to the fact that the SQL provided by the developer only covers part of the full statement (only the FROM and *WHERE* clause) that is sent to the DB.

Considering the above example:

```
SELECT a.number, a.history, a.title, a.text
FROM CONTRACT a
WHERE a.number = ?
and a.history = (select max(history) from CONTRACT c
where c.number  = a.number)
```

the first part:

```
SELECT a.number, a.history, a.title, a.text FROM CONTRACT a
```

is generated by the Fw based upon the base mappings for the Contract class. This part cannot be manipulated as part of the user-defined relationship definition. The second part

```
WHERE a.number = ?
and a.history = (select max(history) from CONTRACT c
where c.number  = a.number)
```

must be provided by the UDR definition in form of an instance of ***MicFwRdbSQLClause***. The definition must contain the following items:

- Alias reservations for table names used in the user-defined relationship clause. These are required in order to avoid conflicting alias assignments between the pregenerated and the user-defined part of the SQL statement.
- SQL clause text. This is the SQL that appears after the "WHERE" keyword
- For example, the following must be done in order to create the above statement:
- assign the alias "a" to table CONTRACT (which is referenced by that name in the subselect)
- reserve the alias "c" (for the second reference to CONTRACT in the subselect).
- give the actual SQL WHERE clause

For dynamically defined user-defined relationships, these steps have to be performed by a program in an instance method for each side of the relationship. The method that answers the SQL clause for Customer>>contract may look like:

```
sqlClauseForContract
"Public - dynamically create and answer the SQL clause for relationship con-
tract"
stream := WriteStream on: ''.
stream
nextPutAll: 'a.number = ';
nextPutAll: self contractNumber printString;
nextPutAll: ' and a.history = (select max(history) from CONTRACT c where c.num-
ber  = a.number)'

^MicFwRdbSQLClause from: #('CONTRACT a' '@ c') where: stream contents
```

Note: The above assumes that a method called *contractNumber* exists in Customer, which returns the number of the associated Contract.

The definition of static user-defined relationships follows the same principles but takes place interactively within the STOPF tool.

3.7.5. Restrictions

Contrary to FKRs, modifications to relationships mapped as user-defined relationships are not automatically propagated to the DBMS. For example, changing a relationship mapped as FKR results in the appropriate UPDATE statements for the foreign key columns. For NM relationships that are mapped to an associative table, changes are reflected as INSERT and DELETE statements for rows in that table. If changes to user-defined relationships must be reflected on the DBMS side, it is the responsibility of the application to ensure that the appropriate modifications occur.

In the above example, CUSTOMER (CONTR_NO) is implicitly used as a partial foreign key and must therefore be updated if the Customer>>contract relationship is changed. To achieve this, the column must be mapped to an instance variable in Customer (we called it *contractNumber*). Now the application must make sure that the variables are always updated in order to reflect the correct values. This can be done by overloading the following methods:

```
!Customer methods !

persistenceInsert: aVersion
"Private - ensure storage of contractNumber"
| a |

(a := contract getLoaded) notNil
  ifTrue: [ self contractNumber: a number.]
  ifFalse: [ self contractNumber: nil.].!

persistenceUpdate: aVersion
"Private - ensure update of contractNumber "

(aVersion hasVersionFor: #contract)
  ifTrue: [aVersion recordNewVersion: self contract number
    for: #contractNumber.]
  ifFalse: [aVersion recordNewVersion: nil
    for: #contractNumber.].! !
```

3.7.6. Class MicFwRdbSQLClause

This class provides the ability to describe an SQL WHERE clause with some syntax extensions. These extensions are:

Inside the WHERE clause, positions which must be replaced by actual values at runtime can be given with a "named parameter syntax". This means that the name of the instance variable of the source object which supplies the value (e.g. '?name') is given after the usual parameter marker ('?'). This means that the Fw will get the parameter value from the relationship source object at runtime by sending it the get message for that variable.

You can also include non-column-mapping variables as parameter in the sql-clause of the user-defined-relationship:

instance variable	Description	Sample
?instVarName	column mapped instance variable	?id
?relationship	for one-column relationship mappings (foreign-key side)	?ownerEmployee
?(relationship index)	for a specific column of a multi-coulmn relationship mapping	?(ownerEmployee 1)
?(instVarName aggregateSelector)	for aggregated instance variables	?(name firstName)
?(SUBTYPE)	the bottom level subtype value for the receiver class	?(SUBTYPE)

Table 8: Non-column-mapping variables for user-defined-relationship

This is mainly used for statically defined UDRs. When you use dynamic UDR's and parameter marker, you

must give the MicFwRdbSqlClause class the *#markToBeCompiled* method to compile the sqlClause explicitly.

Extra table names and aliases for the FROM clause can be defined in the first (*#from:*) parameter. This parameter consists of an array of Strings. The syntax for each String is '<Table> <alias>', where <Table> is a table name that will appear in the FROM clause and <alias> is an alias name assigned to <Table>. If <Table> is the name of a table that is already included in the FROM clause by Fw generation, it will only appear once, but with <alias> assigned to it. <Table> can also be '@', which will not appear as a parameter name in the FROM clause, but <alias> is reserved and will not be used by Fw statement generation. This is used if an alias is assigned inside a subselect.

3.8. Special relationship properties

3.8.1. Cached relationships

3.8.1.1. An example

The Currency class is referenced by many other classes. Only a limited number of instances ever exist, and these instances rarely change. We therefore decide to keep instances of the class in an application-maintained cache (e.g. in a class variable) which is initialized at program startup.

The Account class has a to-1 relationship to Currency in the instance variable #currency. Both classes are mapped to their own tables; the relationship is mapped to a foreign key.

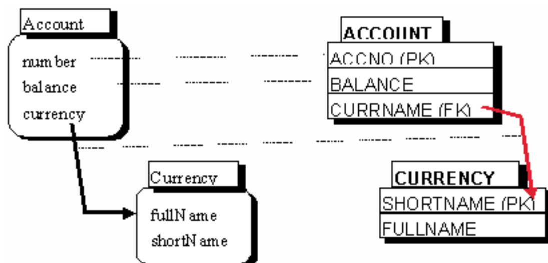


Figure 34: Mappings for Account-Currency Relationship

Now, consider the following sequence of statements:

```
acc := Account where: [:accnt | accnt balance > 100].
Transcript nextPutAll: acc currency printString
```

Normally, as soon as *acc currency* is executed, the objects pointed to by the relationship are loaded from the DBMS. However, since we already have all instances of Currency in our application cache, we do not want to load from the DBMS. Instead, we want to be able to set the relationship to the appropriate objects from the cache.

Again, consider the following sequence of statements which are executed after the above:

```
acc currency: anotherCurrency.
Context commitTransaction.
```

In this case, we want the foreign key columns in the Account table to be updated as usual.

This is achieved by setting the #isCached flag for the relationship *Account>>currency→Currency*.

Note: Cached relationships are always unidirectional.

3.8.1.2. Description

To make a relationship cached, it must be marked with the #isCached flag. This can be done by programming or by using the STOPF tool. The #isCached property is only available for foreign-key based primitive to-1 (unidirectional) relationship mapping.

If a relationship is thus marked, the persistence resolution mechanism will not load the target object for the relationship from the DBMS but instead will send the message *cachedObjects* to the target class (Currency in the above example). This message must be implemented by the application and is expected to return a collection containing instances of that class. The object whose primary key corresponds to the foreign key from ACCOUNT will be selected from the collection. The relationship will be set to that object (or nil if none qualifies).

Relationship updates are treated in the usual way.

3.8.1.3. Implementation

The default caching scheme is to call the method *cachedObjects* on the target class of the relationship. It is the application developers responsibility to implement this method. The method is called by the POM. The resulting collection is searched sequentially based on the foreign key values.

This may get inefficient for classes with many instances. If a more specific caching scheme is desired, this can be implemented by overriding the following method:

```
!MicFwAbstractPersistenceManagerRdb publicMethods !

cachedObjectsFor: abstractTargetClassMapping with: foreignKeys from: depen-
dentMappingInterface
"Answer an instance of @abstractTargetClassMapping mappedClass
whose primary keys join with @foreignKeys for @dependentMappingInterface"
```

```
"** default: delegate to @dependentMappingInterface **"
^dependentMappingInterface
detect: foreignKeys
in: abstractTargetClassMapping mappedClass cachedObjects
for: abstractTargetClassMapping! !
```

Note: @foreignKeys is an array with FK column values. @abstractTargetClassMapping can be used to access the target class through the #mappedClass message.

This way the cache access can be optimized by either organizing and accessing the cache in a more efficient way (e.g. dictionary) or by demand-loading the target object(s).

3.8.2. Unidirectional relationships

Unidirectional relationships do not require a back reference from the target class. Therefore the target class does not have to provide an instance variable for the relationship.

3.8.3. Monitor target

Monitor target is only supported for to-1 relationships. This flag is necessary for the aggregated 1-to-1 relationship, where it will be set automatically when mapping an aggregated relationship. For the other to-1-relationships it is optional. The target object creates a change-event connection to the relationship, so that the other side of the relationship knows, when the target object changed its value

Normally, if a (unidirectional) primitive relationship is mapped, no event connection is created between the relationship proxy and the target object so that changes to the target object can not be detected and propagated to the source object if necessary.

The use of the ***#monitor target*** flag creates the event connection. For a aggregated 1-to-1 relationship, you must set the ***#****monitor target* flag, so that changes to the source object can be detected by the FW.

3.9. Column Mapping Attributes

When mapping an instance variable to a single column, you may have different attributes for that mapping to deal with special database characteristics associated with the column that Framework should pay attention to.

This is possible using either standard *instance variable mapping* as described in chapter Instance Variable to Column Instance Variable to Column, or as part of *Aggregated Relationship Mapping* as described in chapter Aggregated 1-to-1 relationship Aggregated 1-to-1 relationship.

Currently there are three types of attributes:

- The value strategy, which can be separately configured for INSERT and UPDATE
- The reload strategy, which is related to the value strategy and can also be selected separately for INSERT and UPDATE
- The optimistic strategy, which is part of the optimistic concurrency configuration of a POM. Also refer to ‘Optimistic Concurrency’ (page 151).

In most cases, you will use the default attributes. This is an advanced feature of the Framework that offers you utmost flexibility in cooperation between your classes and the database.

Note: ”Write operation" means INSERT or UPDATE operation.

3.9.1. Value Strategy

The *value strategy* defines how the Framework determines the column value to be stored in the database when an UPDATE or INSERT is performed. You can specify the value strategy for INSERT or UPDATE separately for each column mapping. Loading of objects from the database (with an SQL SELECT statement) is not affected by the value strategy.

The following value strategies are currently supported:

- **Instance value.** This is the default strategy. The value to be written to the database is the value of the mapped instance variable of the persistent object.
- **Ignore.** The Framework does not modify the column (when an object insert or update operation occurs). The Framework ignores the value stored in the persistent object. You may choose this strategy when a database trigger, or some other database mechanism will automatically generate the column value.
- **SQL constant.** The framework ignores the value stored in the persistent object and uses the given SQL constant string to provide the column value every time a write operation is performed to that table row. The user may use any valid SQL expression that becomes part of the INSERT or UPDATE statement generated by the Framework. This might be useful when the column value is provided by some database function, e.g. CURRENT TIMESTAMP, DEFAULT, CURRENT DATE etc.
- **Value message.** The Framework ignores the value stored in the persistent object. Every time a write operation is performed to that table row, the Framework sends a unary message to the persistent manager's current CommitHandler instance. The user may use any valid method selector that the CommitHandler of the persistence manager can understand. The value returned by the method which has been called is the value that will be written to the database and into the persistent object. This mechanism can be used for automatic primary key generation or timestamp generation.

3.9.2. Reload Strategy

There might be situations where the current column value stored in the database is unknown to the Smalltalk application immediately after the UPDATE or INSERT statement has been performed. For example, this can occur if a database trigger or SQL constant was used to provide the current column value. In such situations, the persistent object does not contain the current database values after a write operation.

As you can imagine, it is not acceptable for the application programmer to manually refresh or reload such persistent objects. The use of a proper reload strategy solves this problem.

The reload strategy defines if and when a single column value is to be reloaded from the database after a write operation. You can specify the reload strategy separately for INSERT and UPDATE operations. The following reload strategies are possible:

- **no reload strategy.** This is the default. The column will not be reloaded after a write operation.
- **reload after statement.** Every time a write operation affects the table row the Framework will reload the current value from the database after the write operation with an adequate SQL SELECT statement. The reloaded value will be stored in the persistent object.
- **reload after change.** This reload strategy is similar to "reload after statement". The only difference is

that the Framework will reload the column after a write operation that affected the column in the table row. No reload will be performed if the column was not affected by the write operation.

Note: The “reload after change” strategy is equivalent to "reload after statement" when using precompiled statements (e.g. when using Static SQL), because in this case write operations always affect all columns.

Reload strategies are not supported for primary key columns, because they do not make any sense here.

3.9.3. Optimistic Strategy

Setting the optimistic strategy attribute declares a column for use in an optimistic concurrency environment. (See chapter 7: Optimistic Concurrency). The user can choose one of the following optimistic strategies, configurated separately for UPDATE and DELETE operations:

- **no check.** This is the default. The column will not be used for an optimistic concurrency check.
- **check always.** Every time an update operation affects the table row the Framework appends the column to the WHERE clause of the statement.
- **check when changed.** Every time an update operation affects this column the framework appends the column to the WHERE clause of the statement. This option is not available for DELETE operations.

Note: The “check when changed” strategy is equivalent to "check always" when using precompiled statements (e.g. when using Static SQL), because precompiled update operations always affect all columns. Optimistic strategies are not supported for primary key columns, because primary key columns always appear in the SQL where clause to identify the table row of a persistent object.

4

Programming Reference

4.1. Introduction

The programming of database applications - meaning applications which process persistent objects - is supported by the Persistence Framework by means of a series of classes and their public API methods. This chapter gives an overview of the most important programming tasks relating to persistence and shows how the Framework supports the developer during the application programming phase.

4.2. Using a persistence manager

The classes derived from ***MicFwPersistenceManager*** are specially developed for various database systems and database interfaces (ODBC, native DB-connection). The class hierarchy can also be expanded to port the Framework to other database management systems or add application-specific functions such as those for calling database procedures (stored procedures).

The ***MicFwAbstractPersistenceManagerRdb*** class describes the protocol for all relational databases as an abstract class. Methods other than those of a higher, object-oriented level can also be found here, such as methods based on SQL. Persistence managers for object-oriented or network database systems could also be placed at the same level.

The ***MicFwPersistenceManagerOdbc*** class specializes in the ODBC protocol for relational databases. ODBC (Open Database Connectivity) is an important industry standard for application programming which sets the standards for a number of details such as the API, data types, data transfers, error codes, cursor control, etc.

The ***MicFwPersistenceManagerESQL*** class is used for POMs with embedded static SQL facilities for DB2 databases. ESQL is described in detail in ‘Embedded SQL’ (page 163).

The ***MicFwPersistenceManagerOracle*** class implements a POM to access Oracle databases with the AbtDatabase classes of VA Smalltalk. You can find some detailed specifics about using Oracle database in the according passages of this chapter.

4.2.1. Get a global registered POM

In order to use an existing persistence manager which was stored in STOPF using menu item *Manager->Store Globally As...* See chapter ‘Storing and exporting a POM’ (page 186), you can use either ***#named:*** or ***#named:ifAbsent:*** (both class methods of ***MicFwPersistenceManager***). Both API methods answer a POM instance of the receiver class or one of its subclasses, whose names match the parameter which is passed. The ***#named:*** method triggers a ***MicFwSematicError*** if no appropriate POM is registered, whereas ***#named:ifAbsent:*** executes the passed block.

Example

```
...
aPom:= MicFwPersistenceManager named: 'FW Examples'.
...
```

4.2.2. Get a POM from an initializer class

In order to use a POM stored in an initializer class, you can use class methods ***#createPOM*** or ***#globalPOM*** from the initializer class.

In contrast to ***#createPOM***, ***#globalPOM*** first checks whether a POM with the same name has been globally registered, and returns this globally registered POM instance if available. If no registered POM exists under this name, the ***#createPOM*** method is called in order to instantiate the POM from the initializer class and subsequently globally register this new POM instance.

Example

```
...
aPom := MicFwExamplesPomOdbc globalPOM.
...
```

Caution: This is a development-environment feature only.

4.3. Connecting to the database

Before you can work with a database using the Persistence Framework, (i.e. store, load and modify objects), you must create a connection to the respective database system.

This is done using the *#connectTo: dbName user: userid password: password* method defined in ***MicFw-PersistenceManagerRdb***. Since every DBMS defines different database connect protocols, the combination of this method and ***#disconnectDataSource*** serves as a joint protocol for all DBMS. The concrete subclasses of ***MicFwPersistenceMangerRdb*** implement this method specially for a DBMS or database interface (e.g. ODBC).

The concrete subclasses can still provide other database connect methods which make it possible to define other options for the connection. For example, in ***MicFwPersistenceManagerOdbc*** (a POM class for linking the ODBC database interface) there is a method called ***#connectWithDriverPrompt*** that causes the ODBC Manager to open a dialog window for selecting the ODBC data source.

However, you should be aware that the use of database-specific methods has an adverse effect on database independence, i.e. wherever the default behavior implemented in the abstract methods in ***MicFwPersistenceManager[Rdb]*** is adequate, these DBMS-independent methods should be used.

The ***MicFwPersistenceManager>>disconnectDataSource*** method offers a uniform method of breaking the database connection, irrespective of which concrete DBMS or database interface is being used.

In order to check whether a POM is already connected to the database, you can use the ***MicFwPersistence-Manager>>isConnected*** method. This answers true if a valid connection to the database exists, or false if this is not the case.

4.3.1. Configuring the ODBC DLL

Before you can use an ODBC persistence manager under OS/2, note the following:

An ODBC connection can be established in two ways: with the help of a ODBC driver manager or without a driver manager by directly using the target database driver DLL. In the latter case, you must set the appropriate ODBC driver DLL name in your Smalltalk image. This DLL name is ***#ODBCDLL*** by default. This is the correct name when using an ODBC driver manager for OS/2.

In most cases, the database driver DLL should be accessed directly from your Smalltalk environment without any driver manager installed. This is possible if the DLL is compatible to ODBC (e.g. DB/2 and Sybase SQL Anywhere are database systems that provide ODBC compatible driver DLLs). Before a POM can establish a database connection, you must set the name of the ODBC DLL.

When using DB/2, evaluate the following:

```
OdbcObject useDB22CLIDLL
```

When using Sybase SQL Anywhere (the former name was "Watcom"), evaluate the following:

```
OdbcObject useWatcomDLL.
```

When using a driver manager (e.g. from Intersolv Q+E), evaluate the following (this is the default configuration):

```
OdbcObject useOdbcDLL
```

You can also set the DLL name using the ExamplesBrowser tool:

- Open the examples browser (micFrameworks->Browse Examples)
- Select ***MicFwPersistenceOdbcExamples*** (visible when application/package ***MicFwPersistenceOdbcExamples*** is loaded)
- Click on the "Examples" button to switch to category "Tools"
- Select the appropriate method from the list in the top right part of the browser.
- Click on the "Perform" button to evaluate the selected method.

Note: Save your image after setting the ODBC DLL name. This change is activated when the ODBC DLL is accessed for the first time after running your image. If you have attempted to connect to a database previously, exit the Smalltalk image after setting the new DLL name and saving the image. The DLL is used the next time you run the image.

When using the Framework under Windows you need not set the DLL name, because the operating system always has an ODBC driver manager if you have installed ODBC support. Just create an ODBC data source for the database you want to use (in order to obtain a description of how to create an ODBC data source, refer to your operating system documentation or the ODBC support you have installed.)

4.3.2. ODBC connection interface

The information in this section is based on the *ODBTalk* ODBC product and the *Watcom SQL* RDBMS.

Important information is passed to the ODBC driver in the *connect string*. For example:

```
DSN=;UID=;PWD=;Start=dbeng40;DatabaseFile=;
DatabaseName=;DatabaseSwitches=;AutoStop=yes;
KeysInSQLStatistics=;PreventNotCapable=
```

The exact scope of the information passed in the connect string depends on the ODBC driver. The above example is taken from the Watcom environment; for more detailed information, refer to the documentation for the ODBC driver being used.

The connect string is built up from a series of definitions of the form *Name*=*Value,* separated by semi-colons, which assign the connect string attribute *Name* the value *Value*. For example, for the user-ID: *UID=DBA*.

The most important attributes, which must always be set for a database connect, are:

- **UID, PWD.** User-ID and password
- **Start.** A database driver
- **DatabaseFile or DatabaseName.** The database file name (with path) or database name (depending on the database driver)
- **DSN.** Data Source Name.

Additional attributes are optional.

The connect string is created from a data source description stored in the POM. The current value of the connect string can be requested as well as changed:

- ***MicFwPersistenceManagerOdbc>>dataSourceDescription.*** The data source description, which is an object of type ***MicFwOdbcConnectAttributes***.
- ***MicFwOdbcConnectAttributes>>attributeNamed:, MicFwOdbcConnectAttributes>>setAttributeNamed:to:.*** Returns or sets the value of the attribute named.
- ***MicFwOdbcConnectAttributes>>connectString, MicFwOdbcConnectAttributes>> connectStringWith***:. Creates a connect string. When the second form is used, the connect string is modified according to the dictionary that has been passed as a parameter.

And several other methods from ***MicFwOdbcConnectAttributes***.

An ODBC POM also has a few methods for executing the database connect; the database disconnect is always done using ***#disconnectDataSource***.

- ***MicFwPersistenceManagerOdbc>>connectWithoutPrompt, MicFwPersistenceManagerOdbc>> connectWithUser:password:, MicFwPersistenceManagerOdbc>> connectTo:user:password:.*** Establishes the connection to the database without using a prompt dialog from the ODBC driver. In the second form, the user-ID and password can be entered, and in the third form the data source name can be entered.

- ***MicFwPersistenceManagerOdbc>>connectWithDriverPrompt, MicFwPersistenceManagerOdbc>> connectWithDriverComplete.*** Establishes the connection to the database using a prompter dialog from the ODBC driver. The dialog requests all possible attributes (in the first form) or all the necessary, but still unspecified attributes (in the second form). In the first form, the default parameters are already contained in the text boxes.

The database connect for applications that have reached the production stage is always done using ***MicFwPersistenceManagerOdbc>>connectWithUser:password:*** or using ***MicFwPersistenceManagerOdbc>>connectWithoutPrompt***, as no standard dialog window appears when these are used. The application must ensure beforehand that the data source description has been sufficiently specified. For example Watcom SQL.:

```
aPOM := ...
aPOM dataSourceDescription
setAttributeNamed: 'Start' to: 'dbeng40';
setAttributeNamed: 'DatabaseFile' to: 'fwtest.db';
setAttributeNamed: 'DSN' to: 'Test';
[ aPOM
connectWithUser: 'DBA'
password: 'SQL' ]
on: MicFwOdbcConnectionError
do: [ :theException |
Transcript
   nextPutAll: 'Error during database connect: ';
   nextPutAll: theException description;
   cr.
```

```
].
...
aPOM disconnectDataSource.
```

An error occurring during the database connect results in an ***MicFwOdbcConnectionError*** exception. See chapter Error handling: Error handling. Generally, the SQL status from the database driver or the RDBMS provides information as to the cause of the error; the status can be checked using ***MicFwOdbcError>>sql-State***.

4.3.3. ESQL connection interface

For information about how to connect an ESQL POM refer to 'Embedded SQL' (page 163).

4.3.4. Oracle connection interface

Oracle POMs (MicFwPersistenceManagerOracle) provide the connect method:

```
connectTo: dbName user: userid password: password
```

When you are using a local Oracle database, the ***dbName*** **parameter is 2:**

4.4. The programming model

4.4.1. The transaction interface

The transaction interface provides the highest level of abstraction and modest Smalltalk-like handling of persistent objects. In order to do this the transaction facilities of the *Object Behavior Framework* are used and extended. Close integration with the Smalltalk programming model is the reason why only a few persistence-specific method calls are used with the transaction interface beyond handling the transaction context. The following methods make up the transaction interface (note the "get" prefix):

MicFwPersistenceManager:

- ***#getAllInstancesOf:***
- ***#getAllInstancesOf:orderBy:***
- ***#getInstancesOf:where:***
- ***#getInstancesOf:where:orderBy:***
- ***MicFwPersistenceManagerRdb***
- ***#getInstancesOf:whereSQL:***
- ***MicFwPersistenceIdentifier***:
- ***#get***

4.4.1.1. Transaction handling

The basic rule is that all persistency operations are performed within the scope of a persistent transaction context. A persistent transaction context is obtained from a persistence manager (not from the global transaction manager as with the *Object Behavior Framework*). An image transaction is started by sending ***#beginTransaction*** to the context. All changes made within the transaction are buffered until the transaction is committed. The changes from the transaction context buffer are written to the DBMS and committed at the same time.

An important property of the transaction interface is the fact that most operations modifying the persistent state of objects (creation, update) are performed using normal Smalltalk methods and do not require additional explicit messages to be sent.

- A persistent object is created by sending ***#newPersistent***
- Persistent objects are changed using their **accessor** methods
- Persistent objects are deleted using the ***#delete*** method.

In all three cases, the corresponding database operations (SQL INSERT, UPDATE or DELETE) are executed when the transaction is committed. The commit or abort on level 1 of the image transaction always corresponds with a database commit or abort (rollback).

Note: Nested transactions are not supported by the SQL standard nor by any relational DBMS. Therefore the Persistence Framework maps the commit/abort on the final level (level 1) to the corresponding DBMS operation.

4.4.1.1.1. *Deletion Handling*

#delete is only valid within a current persistence context, for both loaded objects and new objects that have been created but not yet stored.

If the #delete message is sent to a persistent instance, this instance is deleted when the next and final commit occurs for the current persistence context. The Framework issues the required SQL DELETE statement against the database.

After a #delete instance has been sent, the following behavior occurs providing that the current persistence context is still running:

Message	Return value
isDeleted	True
isPersistent	True

Table 9: Behavior of deletions (before final commit of the context)

Behavior after final commit of the context:

Message	Return value
isDeleted	False
isPersistent	False

Table 10: Behavior of deletions (after final commit of the context)

As soon as a #delete has been sent for an instance, it is no longer referenced by any relationships, i.e. the instance is removed from all relationships in the image that refer to this instance as a target object. The object net in the image therefore represents the situation after a successful #commit.

IMPORTANT: The deleted instance is only removed from the relationships in the image. The Framework issues no other SQL statements (UPDATE or DELETE)! This may cause the database to report an integrity violation when the transaction is committed if the deleted object is still referenced by foreign keys that have been defined with DELETE RESTRICT.

The application itself is responsible for removing the deleted object from the relationships concerned and for converting the relationships concerned if the underlying foreign keys have not been defined with DELETE CASCADE.

Since the deleted object is no longer referenced by any relationships, all two-sided relationships for the deleted objects are also empty (so that both sides of the relationships show a consistent image.)

Primitive (i.e. one-sided) relationships for the deleted object are not emptied (because they are not affected by the DELETE of the object.)

Each object with a relationship that references the deleted object triggers a change event as soon as the #delete request has been sent (this allows open views to be automatically updated, such as when using the Fws application.)

4.4.1.1.2. Special features of foreign keys with ON DELETE CASCADE

The DELETE rule for foreign keys in the affected relationship that point to an object that is to be deleted are taken into consideration by the Framework. If the image contains instances in which relationships that are affected by the delete have been mapped using a foreign key that has "ON DELETE CASCADE" as the DELETE rule, the Framework carries out the following additional actions:

- The objects that refer to the deleted object with ON DELETE CASCADE are automatically deleted by the database when the database transaction is committed. The instances in the image which the Framework identifies as needing to be deleted when an object is deleted are also marked as deleted and, after the #delete, behave in the same way (see above table) as the object that has been explicitly deleted by the application.

IMPORTANT: In order to ensure that all affected instances are discovered by the Framework at the point in time of the #delete message, all affected **instances should be in the POM object cache at this point in time**. If this is not the case, a multi-level DELETE CASCADE cannot reach all instances in the image. This makes the state of the image inconsistent. The affected relationships do not need to be dissolved, but is advisable because it would speed up performance when finding affected objects.

- The Framework does not issue any explicit SQL Delete statements for objects that are deleted by the database using CASCADE.
- Cascaded deletion affects the object net in the current state at the point in time of the #delete message. This means that:
  - Relationships that have been changed in the current context such that they now reference the deleted object and whose underlying foreign keys still have another value in the database are also deleted during the commit. In order to make this possible the changes for these relationships are written before the cascade is triggered by the DELETE statement.
  - Relationships whose foreign keys still reference the deleted object in the database but have been changed in the current context such that they no longer reference the deleted object are NOT deleted. As in section 3a), this is made possible by writing the changes for these relationships before the DELETE statement.
  - The DELETE cascade is simulated for new objects that have been created and are therefore not yet in the database, i.e. the Framework issues **explicit DELETE statements** for objects that reference this object in the object net.
- The Framework relies on the fact that the database completely deletes instances that are affected by the CASCADE. It is particularly important to take this into consideration if instances that are to be deleted using CASCADE are mapped to more than one table:

- The foreign keys in the internal table connections must be defined with ON DELETE CASCADE. Otherwise the database would report an integrity violation during the #commit. These foreign keys also automatically delete the rows in the dependent tables.
- If a foreign key with the DELETE CASCADE rule is not defined in the master table but in a dependent table, explicit triggers must be defined that ensure that the rows in the master table are also deleted (and the respective supertable). Example:
  - Class Employee mapped to table: EMPLOYEE
  - EMPLOYEE
    - EmpNo (Primary Key)
  - Class Customer mapped to 3 tables: PARTNER, PERSON, CUSTOMER has a relationship to employee, whose foreign key is not in the master table.
  - PARTNER
    - Id (Primary Key)
  - PERSON
    - Id (Primary Key, Foreign Key references PARTNER ON DELETE CASCADE)
  - CUSTOMER
    - Id (Primary Key, Foreign Key references PERSON ON DELETE CASCADE)
    - EmpNo (Foreign Key references EMPLOYEE ON DELETE CASCADE)

In this case a trigger such as the one below must be defined, that also deletes all entries in the PARTNER and PERSON table for the cascaded customer deletion. DB2 Syntax:

```
CREATE TRIGGER Kunde_Loeschen
AFTER DELETE ON KUNDE
REFERENCING OLD AS old_row
FOR EACH ROW MODE DB2SQL
DELETE FROM PARTNER WHERE id = old_row.id
```

Otherwise invalid, inconsistent instances remain in the database and may lead to various problems!

4.4.1.2. Caching

For objects loaded into the image, object identity is maintained by registering them in an object cache which is kept in the persistence manager. The persistence cache is keyed by the primary key values defined for each persistent object class. If an object gets reloaded (either explicitly or implicitly through a relationship), the cache is checked ***after*** the load operation to determine if the object is already in the image. If it is found, the existing object is substituted for the newly loaded one.

4.4.1.2.1. Option Compare

If the ***#optionCompare*** option is specified in the persistence manager, the states of the new and the already loaded objects are also compared. If a difference is found (which means the database and image are out of sync), an exception is signaled.

You can modify the comparison and exception handling of a POM with enabled ***#optionCompare*** **by overwriting one of the following methods in your own POM subclass.**

```
handleCompareCached: cached loaded: loadedObject
"Private - Handle the comparison between a persistent object cached in the
objectCache and the loaded instance from the database"

^(self persistenceCompare: cached to: loadedObject)
   ifTrue: [ cached ]
   ifFalse: [ self handleCompareConflict: cached with: loadedObject ]
```

The ***#handleCompareCached:loaded:*** method is send, when ***#optionCompare*** is true and whenever an object has been resolved from the cache and the cache already contained a different instance.

The method compares both instances with the ***#persistenceCompare:to:*** method whose result decides about whether conflict handling is neccessary or not.

```
handleCompareConflict: cached with: loaded
"A compare conflict has occured. Handle the problem accordingly.
```

Alternatives:

- Reresh the state of the cached object with the state of the loaded object

```
self refresh: cached with: loaded.
   ^cached
```

- Copy the state of the loaded object into the cached object and answer the cached object:

```
self persistenceCopyTo: cached from: loadedObject.
```

```
^cached
```

- Return the cached object without synchronizing (default)

```
^cached
```

This method is send by the first one, when the cached and the loaded instances have different contents. The method must answer the object to be used in the application and is responsible to synchronize the contents of the result. The method comment shows different alternatives how you can modify the implementation.

4.4.1.3. Example

The following shows a typical transaction interface code sequence:

```
...
aPOM := ...
aContext := aPOM newPersistenceContext.
aContext beginTransaction.
aCarCollection := aPOM
getInstancesOf: CarModel
where: [:model | model number = 22].
aCar := aCarCollection isEmpty
ifFalse: [aCarCollection first]
ifTrue: [nil].
aCar notNil ifTrue: [
anAccessories := aCar accessories first.
"edit instance"].
...
aContext commitTransaction.
...
```

The first Accessories instance for the first CarModel instance of the entire set of objects (extent) in the CarModel class is loaded into the cache if it exists, and can be processed further within the framework of the transaction. The changes made in the next lowest transaction level are only adopted when ***MicFwPersistenceContext>>commitTransaction*** is called and are then written into the database if the commit is a level 1 commit.

4.4.2. The direct interface

The direct interface is a more basic alternative to the transaction interface. It does not require a transaction context and therefore may be appropriate where the additional complexity of handling such contexts is not worthwhile. The main differences to the transaction interface are:

- persistence and Smalltalk operations are distinct
- persistence operations are explicitly invoked through a persistence manager
- persistence operations are executed against the DBMS when issued
- object identity is not maintained
- object updates are (currently) not supported

Due to the fact that object identity is not maintained (and the object cache is not accessed), the direct interface incurs less runtime overhead on loading large quantities of objects. Objects that have been loaded via the direct interface can be transferred into transaction processing. See chapter Object state transitions: Object state transitions

The following shows a list containing the most important methods that make up the direct interface. Note the "load" prefix for those methods that retrieve objects from the DBMS, as opposed to the "get" prefix for the equivalent methods in the transaction interface.

MicFwPersistenceManager:

- ***#loadAllInstancesOf:***
- ***#loadInstancesOf:where:***
- ***#loadInstancesOf:where:orderBy:***
- ***#insertObject:***
- ***#deleteObject:***
- ***#deleteAllInstancesOf:***
- ***#deleteInstancesOf:where:***

MicFwPersistenceManagerRdb:

- ***#loadInstancesOf:whereSQL:***

MicFwPersistenceIdentifier :

- ***#load***

MicFwPsProjectionDescriptor:

#load

4.4.3. The SQL Interface

The Persistence Framework also gives direct access to SQL if necessary. The SQL interface allows arbitrary SQL strings to be passed to the Persistence Framework, which are then passed on to the underlying database API. Results are returned as arrays of scalar values. The following methods in class ***MicFwPersistenceManagerRdb*** form the SQL interface:

- ***#executeSQL:***
- ***#executeSQLForFetch:***
- ***#executeSQLForFetch:with:***

The ***MicFwRdbDirectStatement*** class provides similar functionality to the above-mentioned methods. Additionally, instances of ***MicFwRdbDirectStatement*** can be kept in instance variables or used as parameters.

4.5. Accessing objects

4.5.1. Introduction

The way in which an application accesses persistent objects will be described in this section. This includes both read and write access, i.e.

- Inserting new objects / rows of data (*INSERT*)
- Retrieving existing objects (*SELECT*)
- Deleting objects / rows of data (*DELETE*)
- Changing objects / rows of data (*UPDATE*)

When retrieving data, a differentiation is made between *key*-based and *value*-based retrievals. In the transaction interface, objects reconstructed from the database, which are managed in the cache, form the basis for operations that change data, such as deletes and updates. In this section, all *transaction interface* methods are marked with a *(TR)*, and *direct interface* methods are marked with a *(D)*.

4.5.2. Creating objects

There are two class methods that can be used to create new persistent objects: ***MicFwPersistentObject>>new*** and ***MicFwPersistentObject>>newPersistent*** *(TR)*. Normally, ***#newPersistent*** is used right from the start so that the object is placed under the control of the persistence transaction management.

The ***#new*** method creates a "normal" transient Smalltalk object, i.e. this object will not be stored in the database. See chapter Object state transitions: Object state transitions. It can be set to the persistent state later and can be processed in transactions or stored in the database using the ***MicFwPersistenceManager>>insertObject***: *(D)* method. For the latter procedure, it should be ensured that no RI rules are broken - particularly the objects in an object net must be stored in the database in a particular order.

4.5.3. Object retrieval

The retrieval of existing objects refers to the process of searching for certain rows of data in the database, and converting them into Smalltalk objects. From the perspective of the application, retrieval is usually embedded in one of the following procedures:

- **Browser**: A set of objects, the number of which is not known in advance, is returned to the user or the application using the more or less indeterminate search criteria that are given. The objects are evaluated or displayed, possibly so that the user or application can make a more specific selection .
- **Editor**: A specific object is identified and is loaded into the Smalltalk image for editing in a dialog window, for example. The object can be determined directly by means of key-based retrieval using a relationship, or individually selected from the set of objects returned by a browser search.
- **Batch**: A set of objects meeting certain criteria are processed (updated, deleted).

The identity of objects is guaranteed by the transaction interface of the Persistence Framework: the object will be represented internally by its uniquely identifiable instance, based on its primary key attributes, irrespective of which of the methods listed above is used to bring the object into memory. The Framework also contains a series of internal control functions that determine its behavior when conflicts occur (during updates or deletes, for example). The transaction interface methods require an active persistence context at run-time.

Finally, objects can also be migrated between transaction processing and stand-alone processing (without transaction control). In order to show how this works, a list of the direct interface operation and migration functions will be given in this section, including descriptions thereof.

4.5.3.1. Persistence identifier

An object is uniquely identified by the primary key value of an object or row of data. With the help of *persistence identifiers* and the corresponding methods *(TR),* the object can be retrieved from the database. The procedure is as follows:

4.5.3.1.1. Creating a persistence identifier

```
aClass := ..."corresponding class"
aPOM := ... "POM which should be used"
id := aPOM persistenceIdentifierFor: aClass.
```

This results in the creation of an instance of the ***MicFwPsObjectIdentifier*** class.

4.5.3.1.2. Setting the value of the persistence identifier

```
id setValueFor: aSymbol to: aValue.
```

...

With this method, a key attribute identified by the name in *aSymbol* can be set to the value *aValue* in the persistence identifier. It is important that all key attributes are set before actual retrieval takes place, otherwise an error will occur.See 'Error handling' (page 115).

4.5.3.1.3. Retrieving an object

```
instance := id load. "Direct Interface"
```

or

```
instance := id get.  "Transaction Interface"
```

Finally, the uniquely identified persistent object with the specified key attributes can be found in the variable instance.

It is often necessary to reload objects - for example, when the database is being run in read committed isolation mode, which doesn't exclude the possibility that some type of anomaly will arise See 'Setting up a POM' (page 57). Instances can be reloaded by the relevant POM:

4.5.3.1.4. Reloading an object

```
instance := aPOM reloadInstance: instance.
```

When using the read committed and read uncommitted isolation modes, the side-effects described in 'What are concurrency conflicts?' (page 153) can arise: the object may have been deleted (non-repeatable read) or changed in the meantime, leading to the triggering of an exception (see 'Error handling' (page 115)) by the Framework when optimistic concurrency has been set. Otherwise, the application must check for and handle any exceptions.

4.5.3.2. Extent identifier

An extent identifier is used to search for and load objects using a query. The ***MicFwPersistenceManager>>extentIdentifierFor:*** method produces such an extent identifier instance which can be used to create these types of queries.

4.5.3.2.1. Creating an extent identifier

```
extentIdentifier := aPOM extentIdentifierFor: aClass.
```

4.5.3.2.2. Setting the query of the extent identifier

```
extentIdentifier setQuery: aQueryBlock
```

See chapter 'Object Queries' (page 133) for a detailed description of the query syntax.

4.5.3.2.3. Retrieving objects

```
aCollection:= extentIdentifier load. "DirectInterface"
```

or

```
aCollection:= extentIdentifier get. "TransactionInterace"
```

This returns all objects that comply with the extent identifier query in the *aCollection* temp variable.

However, ***MicFwPersistenceManager*** provides an even simpler method of searching for and loading objects using a query:

The ***#getAllInstancesOf:where:*** and ***#loadAllInstancesOf:where***: methods represent this "simplified" method. These methods are implemented exactly as described above. You create an extent identifier, put the passed query in the WHERE parameter and search for/load the objects using ***#load*** or ***#get***.

Examples:

The set of all A-customers:

```
aPOM
getInstancesOf: Customer
where: [:customer | customer vipCode = 'A'].
```

The set of all employees in the marketing department:

```
aPOM
getInstancesOf: Employee
where: [:employee |
   (employee department = 'Marketing') and:
   (employee jobDescription like: '%Clerk%')].
```

The set of all employees with the last name 'Taylor'. The last name is stored in an aggregate object in the instance variable named name:

```
aPOM
getInstancesOf: Employee
where: [:employee | employee name lastName = 'Taylor'].
```

The set of all actors whose last name begins with ‘T’:

```
aPOM
getInstancesOf: Employee
where: [:employee |(employee name lastName like: ‘T%’) and:(employee jobDe-
scription = ‘Actor’)].
```

The class hierarchy can be given as follows, analogous to the example in the tutorial:

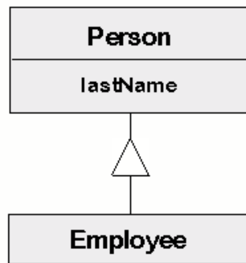


Figure 35: Example of a class hierarchy retrieval

The last request also demonstrates the form of a *subquery*. A subquery is needed when the requested attributes come from different tables. The expressions in the individual subqueries are implicitly connected by ANDs. For details about the Object Query Language (OQL) refer to ‘Object Queries’ (page 133).

In order to load all the instances of a class, you can use the ***#getAllInstancesOf:*** and ***#loadAllInstancesOf:*** methods. Both create an extent identifier with an empty query, i.e. all instances of this class (and all instances of the subclasses) are to be loaded.

Since a retrieval using an extent identifier returns a set of objects, it may be advisable to load the result from the database in a specific order. This is possible using the ***#orderBy:*** method. See chapter: Sorting for relationships and extents Sorting for relationships and extents.

4.5.3.3. SQL extent identifier

In a similar way to extent identifier, the SQL extent identifier is also used to search for and load objects using a query. However, a query based on the objects (and their attributes) is not used for SQL identifier. The objects are searched for in the database using an SQL WHERE clause, which is specified directly.

Since SQL is a standard for relational database systems, SQL extent identifier is only available for ***MicFw-PersistenceManagerRdb***.

4.5.3.3.1. Creating an SQL identifier

```
extentIdentifier := aPOM extentIdentifierSQLFor: aClass.
```

4.5.3.3.2. Setting the SQL identifier query

```
extentIdentifier whereSQL: 'firstName like "H%"'.
```

4.5.3.3.3. Retrieving the objects

```
aCollection:= extentIdentifier load. "DirectInterface"
```

or

```
aCollection:= extentIdentifier get. "TransactionInterace"
```

Objects are loaded in exactly the same way as with the normal extent identifier.

In the SQL extent identifiers WHERE clause, it may be necessary not just to refer to the main table of the object being searched for, but to combine it with other tables. This can be achieved using the ***#from:where:*** method, in which the FROM clause from the SQL SELECT statement is also specified in addition to the SQL WHERE string. See ‘Class MicFwRdbSQLClause’ (page 84).

4.5.3.3.4. ExtentIdentifierSQL with named parameters

SQL-Extent Queries allow the use of named parameters.

Example

```
extent := self pom extentIdentifierSQLFor: MicFwTutorialEmployee.
extent
from: #('@ a' '@ r' 'Employee e' 'Person p')
where: 'p.id in (select r.personid from Residence r, Address a WHERE
r.addressid  = a.id and a.id between ?minid and ?#maxid)'.
extent whereSQL markToBeCompiled.
extent
parameterAt: 'minid' put: 5000001;
```

```
parameterAt: #maxid put: 5000300.
"** load results **"
self addResult: extent load size = 28.
self addResult: extent answerSetSize = 28.
```

Usage

To create a named parameter in the SQLclause of an extentIdentifierSQL, insert the parameter name immediately after the parameter marker (=?).

The SQL Clause allows the following syntax:

```
?parameterName
```

The parameter name will be a string (see example above for parameter 'minid').

```
?#parameterName
```

The parameter name will be a symbol (see example above for parameter #maxid)

```
?(Classname,size,scale)parameterName
```

This syntax provides a type information for the given parameter. <Classname> is the Smalltalk class of the value provided for the parameter named <parameterName>. <size> and <scale> are Integers with the maximum size and scale for the parameter. <size> and <scale> are optional. The default value for <size> and <scale> is <nil>.

Examples:

```
?(Integer)
?(String,10)
?(ScaledDecimal,5,2)
?(ScaledDecimal,,3)
```

Important: When using named parameters with registered ***SQLExtentIdentifiers***, it is **neccesary** to provide an information about the expected type of the parameter value. The framework needs this information to bind the parameter in a correct way to the underlying database API. In the above example this would look like:

```
extent
from: #('@ a' '@ r' 'Employee e' 'Person p')
where: 'p.id in (select r.personid from Residence r, Address a WHERE
r.addressid  = a.id and a.id between ?(Integer)minid and ?(Integer)#maxid)'.
```

The name in brackets must be a valid Smalltalk class name that describes the parameter value type that will be provided through #parameterAt:put:

Performance hints

- The parameter type information is not neccessary when creating a SQLExtentIdenfier dynamically, like this happens in the example. (This would just create performance overhead for the SQLClause parser.)
- You should use Symbols for parameter names when you register the SQLExtentIdentifier in the STOPF tool during development time. Symbols provide a faster lookup in the parameterAt:put: method and you find them easier in your code with 'Browse Senders...' than Strings.
- You should use Strings for parameter names when you create the SQLExtentIdentifier dynamically, because using Symbols would result in a ***String>>asSymbol*** message by the SQLClause parser, which has bad performance and blows up the Smalltalk symbol table.

Caution: Do not forget to send ***#markToBeCompiled*** to the SQLClause (refer to the example above) when the SQLClause contains parameter markers. (The same is with dynamic user defined relationships.) This indicates that the framework must parse the SQLClause and extract the parameter names from the SQL expression. This will not be done by default because of performance reasons!

API Methods

MicFwPsExtentIdentifierSQL>>parameterAt:put:

Provide a value for a named parameter before execution:

MicFwRdbSQLClause>>markToBeCompiled

Set that the SQLCLause needs to be compiled before an SQL statement can be generated, e.g. parameter names will be extracted.

4.5.4. Relationships

The relationships from the Object Behavior Framework also play an important role in the Persistence Framework. In addition to the relationship characteristics described in the Object Behavior manual, the following chapter will describe the special features in connection with the Persistence Framework.

In the Persistence Framework, relationships are resolved on demand, i.e. the objects referenced by a rela-

tionship are not loaded from the database until the contents of the relationship are accessed (e.g. using a *#do:*). Persistent relationships therefore have some additional methods:

- ***#getLoaded.*** Answers all objects which are currently loaded in this relationship.
- ***#removeLoaded.*** Deletes all loaded objects from this relationship.
- ***#loadedSize.*** Answers the number of currently loaded objects for the relationship.
- ***#answerSetSize*** . Perform a SELECT COUNT(*) statement to answer the number of target objects of this relationship from the database. This method is implemented for to-N relationships only.

Of course, all Object Behavior Framework relationship methods are available, but care must be taken, since many methods (such as ***#size***, ***#delete***, etc.) require previous loading of all objects from the database.

4.5.5. Deleting objects

The receiver of a ***MicFwPersistentObject>>delete*** *(TR)* is marked for a persistent delete in the active persistence context. The database *DELETE* is carried out when the *commit* is done in the last transaction level. For details about the behavior of the #delete message, refer to chapter Transaction handling "Deletion Handling ".

The direct interface of the POM also contains several methods of deleting existing objects. One thing they all have in common is the fact that the application is responsible for maintaining referential integrity and consistency in the Smalltalk image (deleted persistent objects shouldn't be left behind in the image).

4.5.5.1. Deleting the extent (all instances of a class) or a part of the extent

- ***MicFwPersistenceManager>>deleteAllInstacesOf:.*** Permanently deletes all instances of the given class.
- ***MicFwPersistenceManager>>deleteInstances:where:.*** Permanently deletes all instances of the given class that fulfill the conditions passed in the query (see value-based retrieval above Object retrieval).

4.5.5.2. Deleting a single object

- ***MicFwPersistenceManager>>deleteObject:.*** Permanently deletes an object. Relationships are not taken into consideration.
- ***MicFwPersistenceManager>>deleteWithReferences:.*** Permanently deletes an object plus all objects that participate in a persistent relationship together with the object.

4.5.6. Changing objects

Persistent changes to the objects in a persistence context must always be carried out by the *Write Accessors* that are generated for the classes, so that changes are recorded within the scope of the transaction handling procedures.

The actual database operations are not performed until the final *commit* in the level 1 transaction. The transaction collects all the changes that have occurred in this persistence context and generates the necessary *UPDATE* in the database.

4.5.7. Application-specific extensions

The Framework provides capability for extending object manipulation functionality in a class-dependent manner. This makes sense when certain operations are absolutely necessary in order to maintain consistency. It could also be done for other reasons such as:

• **Generating primary keys.** A typical method is to call a stored procedure in the database. In the application, the primary key attributes of a new, persistent object are not filled; only after the INSERT in the database do these attributes receive their actual values.

• **Default values.** Sometimes it is useful to fill some of the attributes with certain default values. These values can be constants, calculated values or values generated by the system (the time and date, for example).

• **Replication.** Data fields that lie outside of the mapping (for example, in other table columns or in other tables of the database schema) are dealt with by one of the persistence operations.

• **Consistency.** This category contains all operations that, in one form or another, often update redundant objects or data during persistence operations.

In the Framework, there is a *hook* for each of the persistence operations for implementing the application-specific extensions. The methods used are:

- ***MicFwPersistentObject>>persistenceInitialize***
- ***MicFwPersistentObject>>persistenceInsert:***

- ***MicFwPersistentObject>>persistenceLoaded***
- ***MicFwPersistentObject>>persistenceDelete***
- ***MicFwPersistentObject>>persistenceUpdate:***

These methods are empty and can be re-implemented by the classes derived from ***MicFwPersistentObject***. The *insert* and *update* operations have a *version container,* an instance of class ***MicOvmObjectVersion*** which contains the complete set of information concerning all updates (new values of each of the instance variables) made to the instance.

Examples

Insert extension:

```
Customer>>persistenceInsert: aVersionContainer
"Standard default value and primary key generation"
self vipCode isNil ifTrue: [
self vipCode: ‘C’].
self persistenceManager newPrimeKeyFor: #Customer.
```

Tip: The *newPrimeKeyFor: aClassSymbol* message in the above example is sent to the POM which contains the mapping information and which coordinates and eventually carries out the database operations. In larger applications, it is generally advisable to derive a separate class from the POM class being used to accommodate this specific functionality. In this way, it is possible to update to a new Framework version at any time.

Load extension

```
Person>>persistenceLoaded
"Check and lock for processing later"

self addresses notNil ifTrue: [
self lockFirstAddress ].
```

Update extension

```
Customer>> persistenceUpdate: aVersionContainer
"Supply redundant fields and inform of any important changes"

self updated: TimeStamp current.
(((aVersionContainer oldValueFor: #vipCode) ~= ‘A’)
and: [ (aVersionContainer´newValueFor: vipCode) = ‘A’])
ifTrue: [self notifyNewImportantCustomer ].
```

The ***MicOvmObjectVersion*** class has a number of public methods that serve to analyze and manipulate the update status of the objects whose changes have been recorded (as in ***#newValueFor:***, ***#oldValueFor:*** in the example above). Data manipulation should take place in a disciplined manner so that the entire process remains easy to deal with.

4.6. Iterators

4.6.1. Description

It is often necessary to process the contents of a relationship or an extent one object at a time, without loading the full contents of the answer set into the image. If a relationship is accessed using normal Smalltalk mechanisms (e.g. ***#do:***, ***#select:***, ***#detect:***), it is first resolved from the DB, which involves loading the full answer set into the image. This may exhaust available resources. The same is true for the ***#load*** or ***#get*** methods on ***MicFwPsExtentIdentifier***.

An alternative is the use of a persistence iterator for the relationship/extent. With iterators, each single instance is retrieved from the DB using a FETCH statement. For example:

```
iterator := myPerson addresses asIterator.
[(address := iterator next) notNil ] whileTrue: [
".. do something with the address... "].
iterator release.
"if this is not done explicitly, it will be done when the transaction context
iscommitted or aborted"
```

for extents:

```
extent := pom extentIdentifierFor: Address.
extent setQuery: [:address | address city = 'New York'].
iterator := extent asIterator.
[(address := iterator next) notNil] whileTrue: [
"... do something with address ... "].
iterator release.
```

Note: Fetched objects can be manipulated as usual, e.g. by sending ***#delete*** or by changing attributes using instance variable accessors. However, these changes are written to the DB as usual upon transaction commit. An interface to UPDATE/DELETE WHERE CURRENT is not available at present.

4.6.2. Using iterators

4.6.2.1. Creating an iterator

An iterator can be created by sending the ***#asIterator*** message to an ExtentIdentifier or a PsNRelationship.

Relationship:

```
iterator := customer addresses asIterator.
```

Extent identifier:

```
extent := pom extentIdentifierFor: Address.
extent setQuery: [:address | address addressNumber > 4711].
iterator := extent asIterator.
```

4.6.2.2. Iterators and relationships

The following always fully loads the answer set for the relationship. Then the ***#do:*** iterator is used to process each object. Both approaches are equivalent in this sense:

```
<relationship> asOrderedCollection do: [ :psObj | ... ].
```

or

```
<relationship> getTarget do: [ :psObj | ... ].
```

An iterator is not used!

The following approach uses an iterator to individually load the objects referenced by the relationship from the DB, irrespective of the ***#isLoaded*** property of the relationship. The state of the relationship (***#isLoaded*** and contents) is not changed:

```
<relationship> asIterator allDo: [ :psObj | ... ].
```

Or

```
<relationship> asIterator untilEndDo: [ :psObj | ... ].
```

An Iterator is always used! It will only iterate over objects fetched from the database!

The following approach is the most general and is appropriate in most cases. It is transparent to the application whether the relationship has already been loaded and no DB accesses are performed or a DB cursor is used. If the relationship has already been loaded (***#isLoaded*** = true), the target objects are not loaded from the DB, and the method iterates over the objects already referenced by the relationship.

If the relationship has not been fully loaded previously, the objects are loaded individually from the DB

using a cursor. If the iteration runs to the end of the answer set (without leaving the block), the relationship gets the ***#isLoaded*** property.

```
<relationship> iterate: [ :psObj | ... ]
```

An Iterator is used if the relationship was not loaded! All target objects (also unsaved) are mentioned.

4.6.2.3. Remarks

- Iterators can be used with ***MicFwPsExtentIdentifiers*** or relationships in the same way (using ***#on:***)
- The iterator uses the parameter object persistence manager from the ***#on:*** message.
- Block mode may be implemented using ***#sqlExtendedFetch*** or some form of emulation, depending on the active StatementExecutor.
- An iterator does not change the state of the underlying relationship or extent. The iterator treats the object handed in via ***#on:*** as read-only. Several iterators can be used upon the same relationship/ extent concurrently, provided that the underlying API supports this (e.g. multiple cursor support).
- Objects that are loaded through an iterator are cached in the associated POM (if the transaction interface is active)/
- Static SQL does not currently support opening of the same statement with more than one cursor at one time. This applies to pre-compiled statements (relationships or registered extents) only.

4.6.3. Iteration exceptions

The use of the ***#first***, ***#last*** or ***#previous*** methods requires the StatementExecutor to answer <true> to ***#supportsExtendedFetch***. If an ODBC driver supports the SQLExtendedFetch() function, an appropriate StatementExecutor should be configured. If these conditions are not met a ***MicFwPsSemanticError*** is signaled.

4.6.4. MicFwPsIterator -API

4.6.4.1. Instance methods

4.6.4.1.1. untilEndDo: oneArgumentBlock

Iterate forward starting at the current position. Perform *@oneArgumentBlock* with each persistent object or collection of objects as the parameter. The iterator remains open (and the associated cursor!) and can be reused for other operations (e.g. ***#previous***)

NOTE: This method does not require extended fetch capabilities.

4.6.4.1.2. allDo: oneArgumentBlock

Iterate forward and start at the beginning. Perform *@oneArgumentBlock* with each persistent object or collection of objects as the parameter. The iterator remains open (and the associated cursor!) and can be reused for other operations (e.g. ***#previous***)

CAUTION: This method uses extended fetch, that must be supported by the underlying statement executor!

4.6.4.1.3. next

Single object mode: answer the following object according to the sort order of the underlying extent (arbitrary if no ***#orderBy*** was set). Answer nil if end of answer set was reached

Block mode: Answer an ***OrderedCollection*** containing ***#blockSize*** elements. At the end of the answer set the collection may contain less than ***#blockSize*** (or no) elements.

4.6.4.1.4. previous

Single object mode: Answer the previous object fetched from the persistence medium or nil (beginning of row set)

Block mode: Answer a collection containing fetched objects (maximum size ***#blockSize***) or an empty collection (beginning of row set).

4.6.4.1.5. last

Single object mode: Answer the last object from the answer set of the persistence medium or nil.

Block mode: Answer a collection containing objects (maximum size @***blockSize***) or an empty collection.

4.6.4.1.6. first

Single object mode: Answer the first object from the answer set of the persistence medium or nil.

Block mode: Answer a collection containing objects (maximum size @blockSize) or an empty collection.

4.6.4.1.7. blockSize: anInteger

Set the maximum number of objects (row set size) to be answered by calling ***#next***.

@*anInteger*>= 1 Iterator runs in block mode
@*anInteger*<= 0 Iterator runs in single object mode (this is the default)

4.6.4.1.8. blockSize

Answer the value of instance variable ***#blockSize***.

This method answers 0 when the receiver runs in single object mode

4.6.4.1.9. release

Free all receiver resources.

This is done automatically when ***#sqlAbort*** or ***#sqlCommit*** is triggered by the corresponding persistence manager.

4.6.4.1.10. setSingleObjectMode

Set singleObjectMode. Iterator runs in single object mode (this is the default). ***#next*** answers an object or nil

4.6.4.1.11. isBlockModeActive

Answer true if the receiver runs in block mode, answer false if the receiver runs in single object mode.

4.6.4.1.12. concurrency: rdbConcurConstant

Specify concurrency control for the receiver's cursor. Must be one of the following values in ***MicFwRdbConstants***:

- *CONCURREADONLY.* Cursor is read-only. No updates are allowed (this is the default).
- *CONCURLOCK.* Cursor uses the lowest level of locking sufficient to ensure that the row can be updated.
- *CONCURTIMESTAMP.* Cursor uses optimistic concurrency control, comparing row versions, such as ROWID or TIMESTAMP.
- *CONCURVALUES.* Cursor uses optimistic concurrency control, comparing values.

4.6.4.1.13. cursorType: rdbCursorConstantOrInteger.

An integer value that specifies the cursor type. Must be one of the following values in ***MicFwRdbConstants***:

- *CURSORFORWARDONLY.* The cursor only scrolls forward (this is the default).
- *CURSORSTATIC.* The data in the result set is static.
- *CURSORKEYSETDRIVEN.* The driver saves and uses the keys for the number of rows specified with ***#setKeySetSize:*** option.
- *CURSORDYNAMIC.* The driver only saves and uses the keys for the rows in the row set.

4.6.4.1.14. keySetSize: anInteger.

An integer value that specifies the number of rows in the keyset for a keyset-driven cursor. If the keyset size is 0 (the default), the cursor is fully keyset-driven. If the keyset size is greater than 0, the cursor is mixed (keyset-driven within the keyset and dynamic outside of the keyset). The default keyset size is 0. The key set size must be greater than the rowset size specified with ***#blockSize:***

4.6.4.1.15. cursorType / concurrency / keySetSize

Answer the configured value.

4.6.4.2. Exceptions when setting the modes

The modes can only be set before the iterator is used for the first time (i.e. before the associated database cursor is opened). If the cursor is already open, a ***MicFwPsSemanticError*** is signaled. After a subsequent ***#release*** the modes can be set and will be propagated to the underlying cursor when first used.

Note: Some drivers do not support ODBC 2.0 statement options for ***#cursorType***, ***#keySetSize*** and ***#concurrency*** (e.g. WATCOM).

In such a case it is possible to use the StatementExecutor class ***MicFwRdbStmtExecOdbcWatcom*** which uses the ODBC 1.0 function SQLSetScrollOptions() instead. If a driver does not support a certain option, an exception with a message like "Option value out of range" will result.

4.6.4.2.1. CursorType

If the specified cursor type is not supported by the data source, the driver substitutes a different cursor type and answers SQLSTATE 01S02 (Option value changed). For a mixed or dynamic cursor, the driver substitutes a keyset-driven or static cursor, in that order. For a keyset-driven cursor, the driver substitutes a static cursor. The "Option value changed" error is not signaled since it is not an error.

4.6.4.2.2. KeySetSize

If the specified size exceeds the maximum keyset size, the driver substitutes that size and answers SQL-

STATE 01S02 (Option value changed).

4.6.4.2.3. RowSetSize (BlockSize)

If the specified rowset size exceeds the maximum rowset size supported by the data source, the driver substitutes that value and answers SQLSTATE 01S02 (Option value changed).

For further information please refer to the documentation for the driver and database that are being used.

4.7. Miscellaneous topics

4.7.1. Sorting for relationships and extents

An order can be defined for relationships and extents. This corresponds to the *SortedCollection* facilities in Smalltalk but alternatively to the procedural, block-based interface of *SortedCollection* the frameworks provide a declarative interface. When using this declarative approach, the ordering of persistent relationships and extents has also been applied to cursor-based operations by appending an appropriate ORDER BY clause to the generated SQL statements.

For example, sort criteria can be set using the following statements:

```
Customer addresses orderAscending: #(zip city).
Customer addresses orderDescending: #(zip city).
```

This results in #addresses being sorted in ascending/descending order by zip and city. Both variables must be mapped to columns whose use in ORDER BY clauses is allowed within SQL.

To **order the result by attributes of an aggregated relationship** you can use the following syntax:

```
anAddress persons orderBy: #((name firstName) asc (name lastName) desc).
```

This results in #persons being sorted by their aggregated firstName in ascending and their aggregated lastName in descending order. You must put the selector path of the aggregated attribute in brackets.

When no single aggregation selector is specified, all mapped target selectors are used in the ORDER BY clause:

```
Address persons orderBy: #(name asc).
Customer addresses orderBy: #(zip asc city desc).
```

This results in #addresses being sorted in ascending zip order, and in descending city order therein.

The above applies to **extents** in the same way, i.e.:

```
ext := aPom extentIdentifierFor: Customer.
customers := ext
setQuery: [:customer | customer name like: 'H%'];
orderAscending: #(name);
load
```

will load all customers with names beginning with the letter N, sorted in ascending name order.

Note: Relationships are normally represented internally by ***IdentitySets***, meaning that multiple insertions of identical objects are ignored; the objects only appear once. When sorting is defined for the relationship, the internal representation is changed to a ***SortedCollection***. To keep the relationship consistent the Framework ensures that multiple insertions of identical objects are still ignored although this is not the normal behavior of a ***SortedCollection***.

4.7.1.1. MicFwNRelationship / MicFwPsExtentIdentifier API

4.7.1.1.1. orderAscending: arrayOfSelectors

Define the receiver to be sorted in ascending order of the attributes named in *@arrayOfSelectors*. The attributes are assumed to be base attributes (not relationships)

4.7.1.1.2. *orderDescending*: *arrayOfSelectors*

Define the receiver to be sorted in descending order of the attributes named in *@arrayOfSelectors*. The attributes are assumed to be base attributes (not relationships)

4.7.1.1.3. *orderBy: postfixedSelectors*

Define the receiver to be sorted according to the attributes named in *@arrayOfSelectors*. For each attribute the sorting order can be defined separately by adding ***#asc*** or ***#desc***, e.g. #(#name asc number #desc)

Parameters:

4.7.1.1.3.1. @arrayOfSelectors

An array of selectors corresponding to mapped instance variables. The mappings must be base mappings (direct variable-to-column) that reference columns whose type permits usage in ORDER BY clauses.

4.7.1.1.3.2. @postfixedSelectors

As above. Each selector can also be postfixed with an ***#asc*** or ***#desc*** designator (***#asc*** is default). ***#asc*** causes ascending sort order for the attribute, ***#desc*** causes descending order.

4.7.2. Object state transitions

The methods described in this section bring persistent objects into the Smalltalk image in various ways. Depending on whether a method from the transaction interface or direct interface is used, the objects are either placed under the control of a transaction in the context cache, or are free instances. Making the transition between one of these two states and the transient state is supported by several methods that belong to the transaction interface and which must be called from an active persistence context.

The status of an object is always noted per POM, i.e. if several POMs are being worked with and the POM is changed, the object has the status that the current POM has noted.

4.7.2.1. Transition from the transient to the persistent state

4.7.2.1.1. MicFwPersistentObject>>becomePersistent, MicFwPersistentObject>>becomeNew-Persistent

These practically identical methods transfer the destination instance from the transient state to a state in which they appear to have been newly created in the active persistence context. During the commit phase of the transaction, an insert operation is initiated with the aim of storing the object in the database. These methods are mainly used on objects created outside a persistence context that will, however, be changed within a context. The destination object is inserted into the context cache and registered in the version handler of the persistence context that is currently running. This version handles a management instance that keeps track of changes made to instance variables.

4.7.2.1.2. MicFwPersistentObject>>becomeAllNewPersistent

This method applies the techniques in ***MicFwPersistentObject>>becomeNewPersistent*** recursively to the objects in <u>all</u> relationships.

4.7.2.1.3. MicFwPersistentObject>>becomeLoaded

This method mainly corresponds to the ***MicFwPersistentObject>>becomePersistent*** method mentioned above, except that the destination object does not appear as if it has been newly created. It is mainly used on persistent objects that have come into the system via the direct interface, but are to be changed in a context. The object will eventually find itself in the same state that it would be in if it had come into the context by means of a retrieval or load operation from the transaction interface.

When two instances exist in the image, which are actually 'one' (e.g. the customer with id = 10 was loaded twice times using the direct Interface), a ***MicFwPsObjectIdentityError*** exception is triggered when the application try to put both instances under persistent transaction management using *becomeLoaded*.

4.7.2.1.4. MicFwPersistentObject>>becomeAllLoaded MicFwPersistentObject>>becomeLoadedWith:

This method applies the techniques in ***MicFwPersistentObject>>becomeLoaded*** to the objects in <u>all</u> relationships and, using the second form, any relationships passed to it in the symbol array.

4.7.2.2. Transition from the persistent to the transient state

4.7.2.2.1. MicFwPersistentObject>>becomeTransient

This method releases the destination instance from the context cache. The object is then transient and can be altered outside the persistence context. It can be made persistent again using any of the methods described above.

4.7.3. Default database data types

The ***MicFwDatabase*** class and its subclasses provide information on various database systems and database interfaces. They contain information on data types, mapping between data types and Smalltalk classes, and mapping between data types from concrete databases and database interfaces (such as ODBC, for example).

In order to get useful column typing for a generated DB scheme, you need to specify default database data types for all Smalltalk classes that are used for instance variable typing. This has already been done for the most frequently used basic Smalltalk classes such as ***String***, ***Integer***, ***Date***, etc.. If you want to use other classes (e.g. your own classes) as instance variable types, you should implement a class method that specifies the default database data type for the database interface (e.g. ODBC) you are using.

The class method selector is database-dependent, because each database interface has its own set of data types. Check the ***#defaultDataTypeNamesForClass:*** method of the corresponding database interface class (a subclass of ***MicFwAPIRelationalDatabase***) for the proper class method name.

The method should return a collection of data type names of the target database. The collection should be sorted in data type size order, in order to enable the Framework to choose the smallest data type for the

instance variable type.

4.7.4. Error handling

Whenever an error occurs, either in the Persistence Framework or the database system that is being used, an exception is triggered. A distinction must be made between notifications and errors which indicate that program execution cannot be continued.

4.7.4.1. Persistence Framework Notifications:

4.7.4.1.1. MicFwRdbIntegrityViolation

Is triggered when an integrity violation was detected in the database, e.g. because a primary key that is required is already being used. Attention: if the #skip message is sent, the image may become inconsistent.

4.7.4.1.2. MicFwPsCompareConflict

Only occurs when optimistic concurrency is set. During the re-read of the rows of data that are assembled to create an object, values other than those obtained from the first read may be obtained if an isolation mode is used which is less strict than repeatable read (see also ‘Optimistic Concurrency’ (page 151).

4.7.4.1.3. MicFwPsUpdateConflict

Only occurs when using optimistic concurrency. Updating the object on the database failed because the image timestamps are not valid any more.(see also Optimistic Concurrency)

4.7.4.2. Persistence Framework Errors:

4.7.4.2.1. MicFwOdbcConnectionError

An error has occurred during the ODBC connect.

4.7.4.2.2. MicFwOdbcError

This means that an ODBC error has occurred because of calls from an ODBC to a POM that is not connected to the database, or an internal error occurred during the execution of ODBC statements in the Framework.

4.7.4.2.3. MicFwOdbcStatementError

The ODBC interface has detected an error in the SQL statement that was executed.

4.7.4.2.4. MicFwPsDeletedConflict

Only occurs when optimistic concurrency is set. The rows of data obtained during the re-read of an object may have been deleted in the meantime if an isolation mode is used that is less strict than repeatable read.

4.7.4.2.5. MicFwPsInternalError

This occurs when an unmapped instance variable is referenced by a persistence operation, for example.

4.7.4.2.6. MicFwPsNotStorableError

This exception occurs when a new object is to be stored (inserted), but not all key attributes have been specified, or when an object is to be stored that is marked as being non-persistent. Since this can only occur when the private method ***MicFwPersistentObject>>isPersistent:*** is used, this means that an internal error has occurred.

Tip: this won't occur after applying ***MicFwPersistentObject>>becomeTransient*** because the object is released from the context cache and is no longer marked as having been created or changed.

4.7.4.2.7. MicFwPsSemanticError

Among other things, the following circumstances could be the cause of this type of error:

- There is no longer any connection between the POM and the database
- An attempt is made to update a primary key whose primary key value is referenced by a foreign key
- A persistence operation is carried out without a persistence context or in a "wrong” context (in a normal transaction context, for example)
- A persistence operation is performed without a persistence manager. This occurs when the object has been made transient with ***MicFwPersistentObject>>becomeTransient***
- When an incompletely specified persistence identifier is used during the retrieval of an object.

4.7.4.2.8. MicFwPsUpdateError

This exception is triggered if a ***MicFwPsUpdateConflict*** has not been trapped and processed by an exception handler.

4.7.4.2.9. MicFwRdbIntegrityError

This exception is used if a ***MicFwRdbIntegrityViolation*** has been triggered, but no exception handler has processed the exception.

4.7.4.2.10. MicFwRdbUnknownTypeError

The source of the error is the appearance of an undefined subtype value, i.e. a subtype value which is unknown to the POM. If the subtype column actually contains an incorrect value, then the Framework cannot determine the derived class of the object to be created during the retrieval.

4.7.4.3. Restrictions for mappings to virtual aspects

A virtual aspect is a typable aspect for which no instance variable exists. Virtual aspects can be treated like any other real (physical) aspect when the 4 accessor methods (basicRead, basicWrite, read and write) have been implemented.

There are some effects that have to be taken into account when mapping a virtual aspect: The persistence framework analyses the contents of a persistent context to compute the SQL UPDATE and INSERT statements and their parameter values during #commitTransaction. Therefore the frameworks makes some assumptions about how the changes for the aspects are versioned, e.g. where the changed values are to be found in the ObjectVersions.

Because the accessor methods of virtual aspects are not generated by the framework, it is not safe, that the new (uncommitted) value can be found in the version or that there is a version at all. So it might happen, that the framework does not write changes on virtual aspects to the database during #commitTransaction.

To ensure that the framework will not forget UPDATES for virtual aspects, you must ensure, that the set-accessor (writeAccessor) of the virtual aspect will create an ObjectVersion for the receiver object.

For example this can be implemented like this:

```
self transactionManager writeVersion: anAspect in: self  value: newValue.
```

Where

- self = receiver (the persistent object)
- anAspect = name of the virtual aspect as a Symbol
- newValue = new value (a dummy value used to create the version)

When mapping aggregate or other relationships to virtual aspects, you also have to ensure, that a version of the relationship will be created by write accesses to the target object. (This is the same, that the ‘monitor target‘ flag is used for.)

4.7.5. Empty strings stored in Oracle databases

When writing an empty string into an Oracle database, Oracle does not divide between '' and nil. This was our experience with Oracle 7.3 under Win-NT. So with the next reload, the attribute that contained the empty string contains a <nil> value.

From the Oracle 8 documentation: "Oracle currently treats a character value with a length of zero as null. However, this may not continue to be true in future versions of Oracle."

So the framework cannot distinguish between a <nil> value and a ‘’ value. This will also produce compare conflicts when #optionCompare is enabled. (refer to chapter Caching: Caching)

If you always want to have empty Strings instead of <nil> values, adapt the type converter as described below:

- Subclass class ***MicFwTypeConverter*** (or the type converter you are using)
- Overwrite method ***#fromString:toString:action:truncate:*** so that an empty string will be returned for a <nil> parameter.
- Use this new type converter for all instance variables where you wish to have the modified behavior.
- Open the STOPF tool on your POM and choose Manager→Compile Mappings. Close the STOPF and store the POM globally.
- To ensure, that this new type converter class is used as the default typeconverter (in the future), you could overwrite the method ***#defaultTypeConverter*** in your subclass of ***MicFwPersistentObject*** as well.

4.8. Reentrancy

This chapter discusses thread-safe programming with the persistence frameworks and where are possible reentrancy problems when using the frameworks.

When using the persistence frameworks you run your application on several layers, where the problem of thread-safe programming occurs on each layer in a different shape:

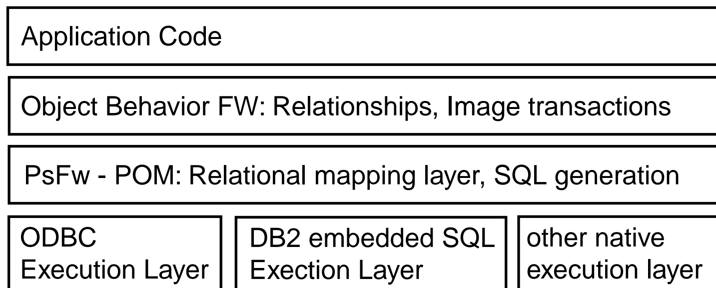


Figure 36: Persistence Frameworks application layers

If a POM is thread-safe or not depends on several factors:

- The underlying database driver must be thread-safe.
- The SQL execution layer or interface code written in Smalltalk must be thread-safe or must be used in a thread-safe way by the frameworks.
- The framework components of the POM must be thread-safe.

When speaking about reentrancy in this document, we speak about the relational mapping layer only. The persistence framework use some 3rd-party SQL execution layers, e.g. Abt-Database classes, Abt-EmbeddedSQL, etc. Whether the database driver, SQL execution or API layers are thread safe or not depends on their implementation and should be clarified in their respective documentation.

You cannot assume that all database drivers are thread-safe. For example, Oracle is writing about its OCI-layer:

> *"[...]While the ODBC driver is thread safe, the OCI layer is not thread safe if multiple threads execute SQL statements in the same connection. Further, only one statement per connection will be simultaneously executed by the Oracle Server. To achieve best performance, applications should use multiple connections with one thread per connection. [...]"*

This seams not to be a problem in Smalltalk, as long as AbtDbmSystem allCallsThreadedPreference == false so that Smalltalk is just one thread for the driver DLL. (This is the default.)

Our tests showed, that ODBTalk (used by ODBC-POMs) is thread-safe.

Framework POM-components that are thread-safe are all components used by Runtime-POMs (RT-POM). The mappings, SQL generation and registry handling of RT-POMs are thread-safe. Some details about this are explained in the passages below.

Note that the "tool-POM" (a POM that can be edited with the STOPF-tool) is ***NOT*** thread-safe.

This can be proven when a push button invokes some code like:

```
[ myPOM getAllInstancesOf: MyMappedClass. ] forkAt: Processor userBackgroundP-
riority.
```

When 'myPOM' is a tool-POM this could cause side-effects or objects with wrong contents in their instance variables.

When 'myPOM' is a RT-POM this is valid code that should not cause unexpected side-effects.

4.8.1. Registered objects

The registry in the POM-core can hold all kinds of objects that implement the Registerable interface. These are:

- OQL-Queries (MicFwPsExtentIdentifier)
- SQL-Queries (MicFwPsExtentIdentifierSQL)
- OQL-Projections (MicFwPsProjectionDescriptor)
- Simple SQL-Statements (MicFwRdbDirectStatement)
- OrderClauses (MicFwRdbOrderClause)

The registry is important when the application uses or plans to use static SQL. Some of these registered objects contain dynamic state or need references to dynamic objects (e.g. the POM) when the application

uses them. (The only exception is OrderClause, because it is 100% static.)

In former releases the use of registered objects was not thread-safe, because the POM returned the registered instance and not a copy. In the current release, the POM returns a copy of the registered object, if neccessary where the dynamic components are copied. So it is possible to use the same registered object for multi-threaded execution, when each thread works with an own copy.

If you access the registered object before execution with the dedicated POM API-methods, such as: #registeredExtentSQLAt:, #registeredExtentAt:, etc. you will get a new copy of the registered object every time.

You can create a thread-safe-copy yourself by sending the registered object the message #preparedForUseIn: <aPOM>. Each registrerable object understands this message.

Correct example source: (Each call to #registeredExtentAt: returns a new copy of the OQL-ExtentIdentifier.)

```
[ | extentIdentifier |
extentIdentifier := aPOM registeredExtentAt: 'myExtent'.
extentIdentifier parameterAt: #myParameterName put: 'parameter value 1'.
resultCollection1 := extentIdentifier load. ] fork.

[ | extentIdentifier |
extentIdentifier := aPOM registeredExtentAt: 'myExtent'.
extentIdentifier parameterAt: #myParameterName put: 'another parameter value'.
resultCollection2 := extentIdentifier load. ] fork.
```

'aPom' is a variable containing the POM in which the extentIdentifier named 'myExtent' was registered.

#myParameterName is the name of a query variable used in the OQL-expression of the extentIdentifier named 'myExtent'.

'resultCollection1' and 'resultCollection2' are variables containg the result of the load operation of the extentIdentifier, where the parameter value is different in both executions.

The following is also **correct**: (The copy of the extentIdentifier is created explicitly in the example source)

```
| extentIdentifier |
extentIdentifier := aPOM registeredExtentAt: 'myExtent'.

[ | ownCopy |
ownCopy := extentIdentifier prepareForUseIn: aPOM.
ownCopy parameterAt: #myParameterName put: 'another parameter value'.
resultCollection2 := ownCopy load. ] fork.

extentIdentifier parameterAt: #myParameterName put: 'parameter value 1'.
  resultCollection1 := extentIdentifier load.
```

This is **incorrect** because both processes work on the identical registered object:

```
| extentIdentifier |
extentIdentifier := aPOM registeredExtentAt: 'myExtent'.

[ extentIdentifier parameterAt: #myParameterName put: 'parameter value 1'.
resultCollection1 := extentIdentifier load. ] fork.

[ extentIdentifier parameterAt: #myParameterName put: 'another parameter
value'.
resultCollection2 := extentIdentifier load. ] fork.
```

4.8.2. Iterators and Multi-Threading

When using a persistence iterator, a database cursor stays open for subsequent fetches of objects until the iterator is closed.

Persistent Iterators in the current release are not thread-safe.

This means, that you cannot use the same iterator in multi-threaded processing.

The following example code is **incorrect:**

```
| iterator ready |
iterator := aPerson addresses asIterator.
ready := 0.

[ iterator untilEndDo: [ :anAddress | anAddress doSomething. ].
ready := ready + 1.
```

```
ready == 2 ifTrue: [ iterator release ].
] fork.
[ iterator untilEndDo: [ :anAddress | anAddress doSomething. ].
ready := ready + 1.
ready == 2 ifTrue: [ iterator release ].
] fork.
```

Using the same iterator multi-threaded can cause errors while reading objects or read objects can have wrong contents in their instance variables.

You can use a separate iterator for each process. The following would be **correct**:

```
[ | iterator |
iterator := aPerson addresses asIterator.
iterator untilEndDo: [ :anAddress | anAddress doSomething. ].
iterator release.
] fork.

[ | iterator |
iterator := aPerson addresses asIterator.
iterator untilEndDo: [ :anAddress | anAddress doSomething. ].
iterator release.
] fork.
```

5

Run-time POM

5.1. Introduction

5.1.1. What is an RT-POM

The RT-POM is not code-generation based but instance based just like the generic POM ("tool-POM") that is editable with the STOPF tool and other framework tools. The difference to the tool-POM is, that the mapping information of the RT-POM is much more compact and optimized. The mapping information includes only the components that are neccessary for the persistence framework's runtime components, while the "tool-POM" holds all mapping information in a redundant way, that allows the framework tools to modify and browse the POM.

The RT-POM instance is about 1/3 or less of the memory size of the "tool-POM" instance!

The RT-POM does not use any generated code during runtime and is fully instance based. Because of the redesign of all mapping classes and algorithms of the RT-POM its performance is nearly as good as the performance of a POM that used the former OptimizedPomSupport.

5.1.2. Why is an RT-POM necessary?

- The RT-POM offers the same functionality, interface and behavior than the common POM instance ("tool-POM"), but it is doing the same job faster and the instance is smaller.
- The RT-POM's mapping are reentrants while the "tool-POM"'s are not.
- In future releases, the "tool-POM" might not support new runtime-features or will not be runnable at all, because it is mainly used for the FW-tools and some testing that for final runtime packaging.
- Reduced runtime overhead (memory usage). A runtime-POM instance contains smaller mapping instances than the tool-POM.

5.2. Requirements / restrictions / compatibility

5.2.1. Required maps

The Runtime-POM (RT-POM) solution is implemented in the following applications, that are part of the frameworks since release R3.4 V1.1:

- MicFwRdbRTMappingInfos. Necessary runtime classes for RT-POMs (includes RT-mappings)
- MicFwRdbRTMappingInfosInit. Initialization methods for RT-POMs (mapping initialization methods). This application is not neccessary during runtime.
- MicFwRdbRTMappingInfosD. Developlement classes and methods. (includes the new RT-POM generator)
- MicFwRdbRTMappingInfosTools. GUI Tools (includes the new RT-POM generator browser and GUI)

5.2.2. Requirements for application compatibility between Tool- and RT-POMs

When a project is planning to use a runtime-POM in the application delivered to the customer, there are some principles that should be considered early enough to prevent changes on the application.

The internal structure of a runtime-POM differs from the structure of a generic POM. The public interface (micAPI Methods) of both are compatible. So the application developers must be cautious to use or modify (by subclassing) micAPI-methods only. Otherwise there is quite a big risk, that the changes cause problems with the runtime-POM or are without effect.

The following list shows the principles that help you write applications that seamlessly can work with a runtime-POM as well as with the generic POM:

- no direct access of internal structures, mappings of the generic POM
- no use of POM methods that are not categorized as 'micAPI'
- no subclassing of internal classes or overwriting/modification of methods that are not categorized as 'micAPI'

Remember: Generated RT-POM instances are not garanteed to be compatible with newer Framework releases. It might be neccessary to regenerate the RT-POM from a former version.

5.2.3. Requirements for using an RT-POM in an application

5.2.3.1. Use prerequisites to RT-POM applications instead of "tool-POM" applications.

In former releases, your application probably had a prerequisite to application "MicFwPersistenceRdb", when you did not use a runtime POM already. This application contains the "tool-POM" mappings, so you should delete this prerequisite.

When you have used the former Optimized-Pom-Support, your application had a prerequisite to "MicFwRdbOptimizedPomSupport". This application is not supported anymore, so you must delete this prerequisite.

The correct prerequisite for your application, that works with a RT-POM is

`MicFwRdbRTMappingInfos`

and - of course - the prerequisite to the database API that you are using, as there are:

- **MicFwPersistenceOdbc (MicFwStmtExecOdbcDB22, MicFwStmtExecOdbcOracle, MicFwStmtExecInterbase or MicFwStmtExecWatcom)**
- **MicFwPersistenceOracle**
- MicFwPersistenceESQL.

Subclasses/modifications of "tool-POM" classes.

Note that subclasses of classes that are defined in MicFwPersistenceRdb (e.g. MicFwRdbClassMapping) can cause problems for the RT-POM generator, when the "tool-POM" uses them. All modifications of micInternal methods in the mapping classes or subclasses can cause problems for the RT-POM generator.

The RT-POM uses different mapping classes. So when your application is accessing the mappings of the pom via internal methods, these might not be compatible with the mappings used by the RT-POM.

Packaging a Runtime-Image

You can package the RT-POM instance into your target runtime image or you can use a dump file created with the object dumper. It is recommended to package the RT-POM instance into the target image.

5.2.3.2. Restrictions

While the ‘cached‘ attribute for ForeignKey-based to-1 relationship mappings can be toggled during runt-

ime when using a “tool-POM”, this is not possible with a RT-POM. The reason is, that the “tool-POM” uses a MicFwRdbFKDependentMapping in any case for a cached or non-cached relationship-mapping, while the RT-POM has own mapping classes for cached and non-cached relationship-mappings because of optimization reasons.

5.3. Generate Run-time POM Options dialog

5.3.1. Options dialog

In the **System Transcript**: Select **micFrameworks** / **Persistence Tools / Runtime POM Generator**.. The **Choose Persistence Manager** dialog appears.

Double-click on a **POM** in the list. The dialog **Generate Run-time POM Options Dialog** appears:

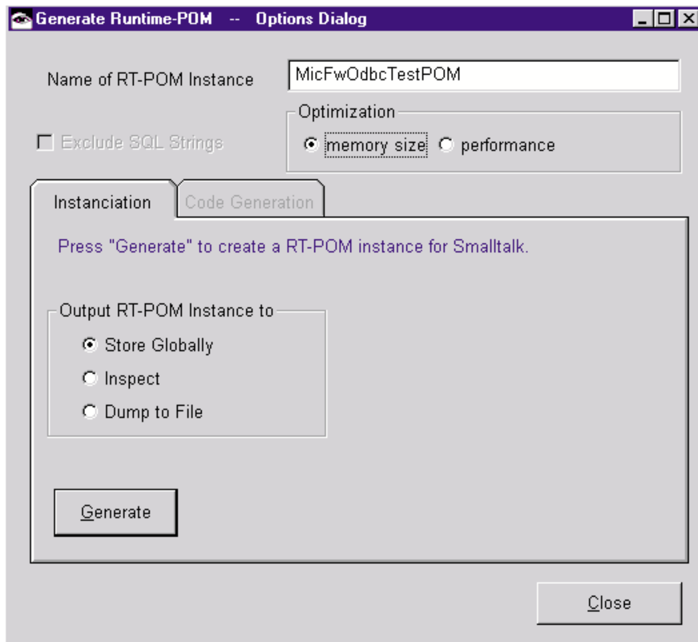


Figure 37. Generate Run-time POM Options dialog

5.3.1.1. Entry field "Name of RT-POM instance"

An entry field with the name that your RT-POM shall have. The default value shown when the dialog opens is the name of the "tool-POM".

5.3.1.2. Checkbox "Exclude SQL Strings"

The RT-POM will contain all statements that are pregenerated in the "tool-POM". Some POMs, e.g. ESQL (Embedded SQL) POMs do not need the SQL-Statement-Strings to execute the SQL statement. When an ESQL-POM is used in static mode (static SQL execution), only the QueryIDs are used to communicate with the database API.

This checkbox is enabled, when the chosen "tool-POM" might not need the SQL-Strings (e.g. when it is an ESQL-POM).

This is a checkbox that determines if SQL-Statement-Strings of pregenerated statements in the "tool-POM" shall be droped in the RT-POM or not.

If this is checked, than the RT-POM will not contain the SQL-Statement-Strings, but only the statement descriptors with their QueryID, that is neccessary for execution of the statements. In some cases this will save a huge amount of memory.

Uncheck this checkbox, when you are using a database API, that needs the SQL-Strings (e.g. when using an ESQL-POM in dynamic connection mode).

5.3.1.3. GroupBox Optimization

5.3.1.3.1. "memory size"

This gives a hint to the RT-POM generator that is is more important to save memory, this means to create a small RT-POM instance than to create a fully initialized RT-POM instance.

5.3.1.3.2. "performance"

This gives a hint to the RT-POM generator that RT-POM shall be as performant as possible even if this would result in a larger use of memory.

In the current release the Optimization is ignored by the RT-POM generator and has no effect!

5.3.2. Tab Instanciation

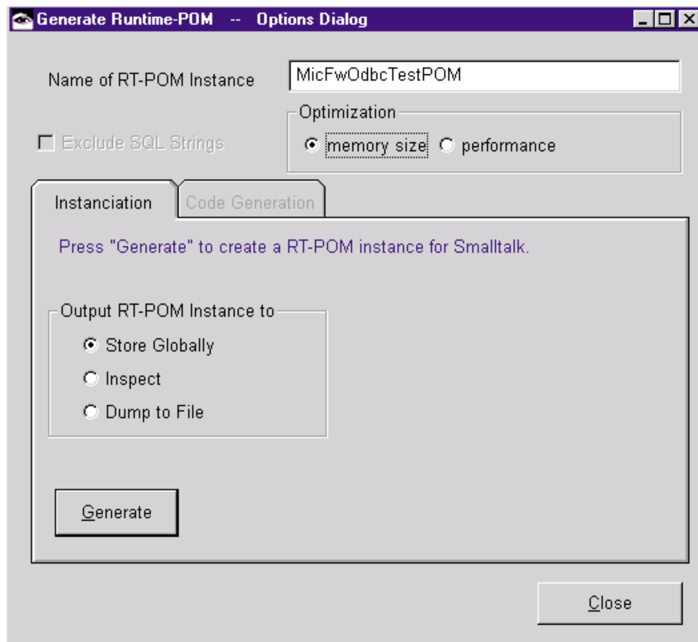


Figure 38. Generate Run-time POM Options dialog tab Instanciation

5.3.2.1. Group box: Output RT-POM Instance to

The RT-POM generator creates an instance of a RT-POM as its output. The RT-POM generator needs a configuration how to return the created instance to you - the user, so that you can use the RT-POM in your application.

5.3.2.1.1. Store Globally

When choosing this option the RT-POM generator sends the "storeGlobally" message to the RT-POM after the generation process. This stores the RT-POM in the POM-classInstance dictionary, so that you can access the RT-POM with the API-method

```
<pomclass> named: <instance name>
```

When your RT-POM has the same name as the "tool-POM" (which is the default), then the RT-POM will substitute the "tool-POM" in the classInstance dictionary.

5.3.2.1.2. Inspect

When chosing this option the RT-POM generator sends the "inspect" message to the RT-POM after the generation process. You get an Inspector-Window with the RT-POM, so that you can decide where you want to keep the instance. (Maybe your application expects the POM in some other class variable or dump file.)

After you have configurated the RT-POM in the dialog, you press the "Generate" button. A progress bar opens during RT-POM generation.

When the RT-POM generator successfully completes the RT-POM generation, you get a message box or inspector. You can close the options dialog with the "Close" button.

You can use the RT-POM instead of the "tool-POM" in your application without changing anything else in your code.

Note: The RT-POM instance has the same StatementExecutor (= database connection) as the "tool-POM". So if your "tool-POM" was connected to the data source before generating a RT-POM, the RT-POM is also connected.

5.3.2.1.3. Dump to file

If selected: Clicking on **Generate** opens a file selection prompt. The generated RT-POM will be written to the specified file (if the file already exists, it will be overwritten).

5.3.2.2. Push button Generate

Click on the push button **Generate** to generate the RT-POM.

5.3.3. Tab Code Generation

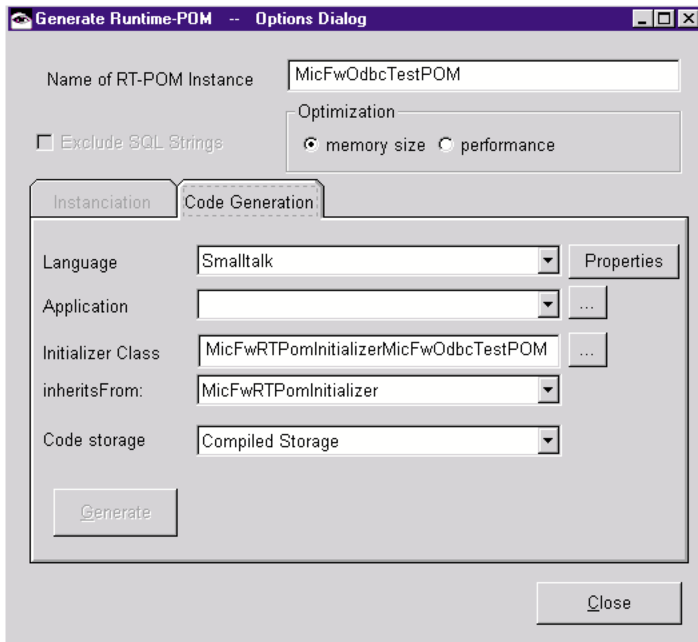


Figure 39. Generate Run-time POM Options dialog tab Instanciation tab Code Generation

5.3.3.1. "Language" Drop-down list

Select the target programming language. Currently only Smalltalk is supported.

5.3.3.2. "Properties" Push Button

Press this button to enable / disable compile time evaluation code. (Code generated for a passive image must not include compile time evaluation expressions.)

5.3.3.3. "Application" Combo Box

Select the target application where the initializer class should be generated. To search for an application that has correct prerequisites, press the "..." button behind the combo box.

The application must have prerequisites neccessary to subclass the class entered in the "inheritsFrom:" drop down list. By default the minimum prerequisite is MicFwRdbRTMappingInfosInit.

5.3.3.4. "Initializer Class" Entry Field

Enter the name of the initializer class to be generated. You can select an existing initializer class with the "..." button behind this entry field.

Remember: Existing methods or classes will stay in their application and will NOT be moved to the given application. Only new methods will be generated to the application given in this dialog, with would extend the initializer class.

5.3.3.5. "inheritsFrom:" Drop Down List

Choose the super class of the initializer class. If the initializer class exists, the superclass will be changed to the super class entered here.

5.3.3.6. "Code Storage" Drop Down List

Choose how the generated code should be written.

You can choose "Compiled Storage", which saves the code into your VA-Envy-repository or you can select "Fileout storage" which will save the code into a text file in chunk format, so that you can install the code with "File In" manually.

It is recommended to use "Compiled Storage".

5.3.3.7. "Generate" Button

This button is enabled when all neccessary information has been entered in the page. After pressing "Generate" the code generation starts immediately.

When code generation is finished, a work space opens that gives information about the number of generated methods, problems and errors that occured during code generation and a description how to use the generated RT-POM when the generation process was successful.

5.3.3.8. Push button Generate

Click on the push button **Generate** to generate the RT-POM.

5.4. RT-POM initializer class

5.4.1. Overview

RT-POM initializer class makes it more flexible and easy to use the RT-POM in your packaged application.

Without a RT-POM initializer class you had two possibilities:

- Transfer the RT-POM instance to packaged image with the VA-packager
- Application could load the RT-POM from a dump file

With a RT-POM Initializer you can:

- Create the RT-POM Instance quickly within your application.
- Distribute a RT-POM save and quickly between other developers in the team.

5.4.2. Creating

To create the RT-POM instance from the RT-POM initializer, create a new instance of the initializer and send it either the message #globalPom or #createPOM.

Example

```
MicFwRTPomInitializerMicFwOdbcTestPOM new globalPom.
```

or

```
MicFwRTPomInitializerMicFwOdbcTestPOM new createPOM.
```

#globalPom returns the globally stored POM instance (retrievable with #named:) or creates and stores it global if neccessary.

#createPOM creates a new RT-POM instance from the receiver and returns it without storing it globally.

5.5. RT-POM and Runtime-Extended Descriptions

5.5.1. Benefits of RT-POM and RT-Extended descriptions

You can convert extended Descriptions to runtime-extended-Descriptions to reduce image size (especially with the additional reduce-option in the extended description converter).

NOTE: After you have converted extended descriptions to runtime-extended-descriptions, you cannot use FW-tools that operate on the extended description, including:

- Object Net Browser
- Object Model Explorer
- POM Generator
- STOPF

Furthermore you can (and should) use a RT-POM instead of a tool-POM, because the RT-POM offers better performance, reentrancy with lower memory consumption.

5.5.2. RT POM runnable with complete or RT extended descriptions

The RT-POM holds references to the attributeDescriptions stored in the extended descriptions, so to make sure, that the RT-POM references the runtime-extended-description's attributeDescriptions and to make sure that the full-blown-attributeDescriptions are no longer referenced, you should create a RT-POM AFTER you have converted extended descriptions to runtime extended descriptions.

You can do that, because the RT-Pom-Generator is runnable with both complete or runtime extended descriptions in the image.

5.5.3. Method 1: Creating a POM out of a POM Storage (Initializer) Class or Export file

The following describes one way of performing the steps to reduce the image size before packaging the runtime image.

5.5.3.1. Remove extended descriptions in image

If you are not sure about the state of the extended descriptions in your image, remove them first. To remove the extended descriptions:

- Select from the **System Transcript**: **micFrameworks / Object Behavior Tools / Convert Extended Descriptions**. A Browser opens.
- Select the radio button **Remove Extended Descriptions** or **Convert to complete descriptions** is selected.
- Press **Start**.
- Close the MessageBox and Browser afterwards.

5.5.3.2. Create Tool-POM instance from storage class or export file

To create a Tool-POM instance from the export-file or storage class and store it globally:

- Select from the **System Transcript**: **micFrameworks / Browse Persistence Manager**
- Choose your **Initializer-Class**.
- Press **Choose**. The STOPF Tool opens.
- In STOPF: Select **Manager / Store Globally As**.
- Confirm the **POM name**.
- Close the STOPF-tool.

5.5.3.3. Convert extended descriptions to runtime extended descriptions

To convert the extended descriptions to runtime extended descriptions:

- Select from the **System Transcript**: **micFrameworks / Object Behavior Tools / Convert Extended Descriptions**. A Browser opens.
- Select **Convert to Runtime Descriptions**.
- Check the check box **Reduce Extended Descriptions**.
- Click **Start**. A message box opens.
- Click **OK**.
- Close the browser by selecting the **Close**.

5.5.3.4. Create RT POM from Tool POM

To create the RT-POM from the Tool POM:

- Select from the **System Transcript**: **micFrameworks / Persistence Tools / Runtime POM Generator**.
- Choose the tool-POM instance that you stored globally above.
- Click **Choose**. The RT-POM-Generator opens.
- Select the destination location for the RT-POM.
- Click **Generate.**
- After the generation process has finished: Close the browser.
- Package the runtime-image.

5.5.4. Method 2: Creating a POM out of a POM Storage (Initializer) Class or Export file

This method is similar but somewhat easier than the method above.

5.5.4.1. Remove extended descriptions in image

As above.

5.5.4.2. Create RT POM from Tool POM

Create the RT-POM from the tool-POM or its initializer class and store it globally.

5.5.4.3. Convert extended descriptions to runtime extended descriptions

Convert all extended descriptions to runtime-extended descriptions (see above).

5.5.4.4. Relocalize the RT-POM

Relocalize RT-POM re-establishes the extended description references in the RT-POM instance to extended descriptions of the classes in the image.

- Select from the **System Transcript**: **micFrameworks / Persistence Tools / Relocalize RT-POM**.
- Select the RT-POM created earlier.

5.5.5. Notes

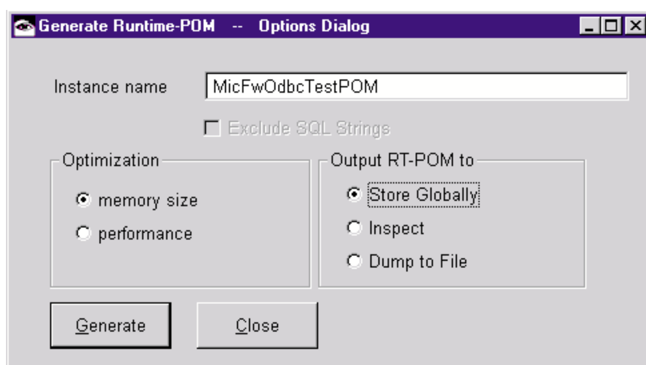
Note 1: You should delete the tool-POM from your image before packaging, if the RT-POM-Generator did not replace the tool-POM with the RT-POM. The RT-Pom-Generator normally overwrites the tool-POM when you have choosen Output "Store globally" in the RT-POM-Generator and when the RT-POM has the same instance name as the tool-POM. (this is the default)

Note 2: When you create the RT-POM BEFORE converting complete extended descriptions to runtime-extended-descriptions the RT-POM will reference the complete-extended-description's attributeDescriptions. This is a memory overhead, that can should be avoided with the steps described above.

Note 3: When you dump the RT-POM into a dump-file, the object-loader will load the attributeDescriptions referenced by the RT-POM together with the RT-POM itself. So whenever you load a RT-POM with the object-dumper, you get a memory-overhead because of those attributeDescriptions that now exists at least twice:

- in the RT-POM, because they have been dumped together with the POM and have been loaded
- in the extended description of the classes.

To get rid of those duplicate references, you should call the method POM>>#relocalizeAspectDescriptions after you have loaded the RT-POM from a dump-file. (This method requires that application MicFwRdbRT-MappingInfosInit is included in your runtime-Image).



6

Object Queries

6.1. Introduction

6.1.1. Purpose of the query interface

No Persistency Framework can be deemed complete without an appropriate query facility. The Mynd Frameworks provide a sophisticated OQL (Object Query Language) implementation which can be used to formulate arbitrary query expressions in a Smalltalk-like syntax. These expressions are translated into SQL by the Framework and executed on the RDBMS with query results returned in the form of Smalltalk objects. In summary, OQL provides the following advantages:

- A database query is formulated in pure Smalltalk. No knowledge of SQL or the underlying DBMS schema is required
- Queries can be parameterized with Smalltalk objects
- Similar to ODMG OQL

In order to satisfy different programming needs, the Persistency Framework provides 3 ways of issuing OQL queries:

- Block-based queries. This interface is best suited for coding queries that are fully known at programming time. It is similar to the well known Smalltalk *select:, detect:* blocks.
- Array-based queries. This is an alternative to block-based queries which is provided mainly for backwards compatibility
- Stream-based queries. This interface is designed to support queries that are dynamically built at runtime.

6.1.2. An example query

Let Person be a persistent class mapped in a persistency manager (POM). The following query loads all instances of Person who are older than 18 and live in "Paris" from the database.

```
classExtent := pom extentIdentifierFor: Person.
classExtent
  setQuery: [:aPerson |
    aPerson age > 18
    and: (aPerson address cityName = 'Paris')]
^classExtent load
```

This executes the query and returns a collection of Person objects.

Queries are issued through an extent identifier (class ***MicFwPsExtentIdentifier***) which is obtained from a POM for a specific class.

The query itself is contained in the block which is passed to the extent identifier as an argument to the ***#setQuery:*** method. The query block looks like any Smalltalk block, and in fact technically it is just that.

However, the way in which this block is evaluated differs from block expressions as in ***Collection>>select:***. The main difference is that the block is not evaluated for Person objects which are already loaded in the image, but it is parsed and translated into SQL. The SQL is then executed in the DBMS and the result is returned in form of Person objects. This difference leads to a few deviations from standard Smalltalk semantics inside the block which are explained in the following chapters.

6.2. The Query Block

6.2.1. Structure

The query block comprises the following components:

- One or two block argument variables. The first variable represents an element of the result set (similar to the block argument in *Collection>>do:*), which represents an instance of the class for which the query extent was obtained. The second block variable represents the query environment. The query environment can be used to access different services within the query block (e.g. definition of variables).
- The definition and declaration of variables that are required in the query expression. The definition obtains the variable from the query environment. The declaration assigns persistent semantics to the variable by using the *#in:* method.
- The query expression itself

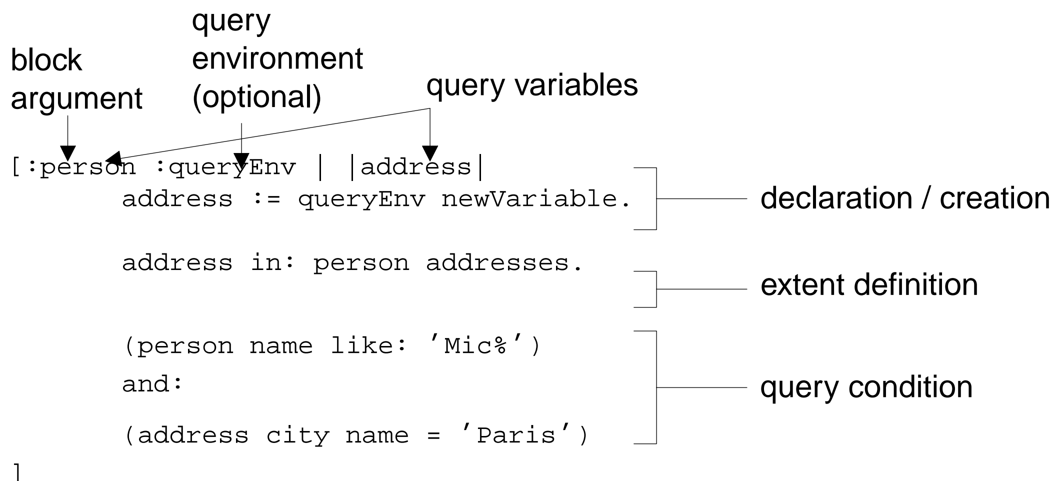


Figure 40: Object Query Components

6.2.2. Query variables

A query variable is:

- the first argument variable (called *result variable*) in the block (e.g. person in the above query).
- local block variables that are defined using *#newVariable*

Query variables represent persistent objects outside the Smalltalk image. The query condition uses these variables by translating the declaration to the appropriate SQL expressions which appear in the final SQL statement.

The result variable is implicitly declared as the entire persistent extent of the class to which the extent identifier relates.

Other variables are declared by either of the following:

- Path expressions on previously declared variables (e.g., the result variable).
- Extent expressions.

6.2.3. Path Expressions

A path expression consists of an arbitrary number of consecutive messages that correspond to persistently mapped relationships. On the far left side of the path expression must be a query variable. Path expressions can appear in query declarations and in the query expression. Example:

```
extent := pom extentIdentifierFor: Customer.
extent setQuery: [:customer :queryEnv |
   | employee address |
"define 2 query variables"
address := queryEnv newVariable.
employee := queryEnv newVariable.

"declare variable1 based on the result variable"
employee in: customer ownerEmployee.
```

```
"declare variable2 based on variable1"
address in: employee addresses.
"query condition"
  employee name firstName = 'Henry'
  and: (address cityName = 'Paris')
].
```

In this query, *employee* is defined by sending the *#ownerEmployee* message to the result variable. *address* is defined by sending the *#addresses* message to the previously declared *employee* variable. The query expression returns all Customers that belong to an employee with firstName "Henry" who also has an address in Paris.

Note the fact that the cardinality of a message does not matter. In the above query, the message *#cityName* is sent to the variable *address*, which is defined on a to-N relationship (*employee addresses*). Therefore one could expect that it contains a collection of Addresses. However, in the context of query variable declaration and query expression, collection types are treated as if they where block argument variables in a nested *#detect:ifNone:* block. In other words, the above query, written in standard Smalltalk (without persistence), would look something like:

```
allCustomers select: [:eachCustomer |
eachCustomer ownerEmployee name firstName = 'Henry'
and: [(eachCustomer ownerEmployee addresses
detect: [:eachAddress |
eachAddress cityName = 'Paris']
ifNone: [nil]) notNil]
```

6.2.4. Extent Expressions

An extent expression is used in query variable declarations only. It declares a variable to be equivalent to the collection of all persistent instances of a specific class. Example:

```
extent := pom extentIdentifierFor: MicFwTutorialAddress.
extent setQuery: [:address :queryEnv | | allCustomers |
allCustomers := queryEnv newVariable.
allCustomers in: MicFwTutorialCustomer.
(address cityName = 'Paris') and:
(allCustomers has: address persons)]
```

The extent declaration makes *persons* a placeholder for all persistent Customer instances. The query expression retrieves all addresses in Paris with at least one Customer living there (*#has:* operator).

Extents are mainly used to restrict queries to instances of a specific class.

ExtentIdentifiers can be combined with the methods:

- union: anExtentIdentifier
- intersect: anExtentIdentifier
- except: anExtentIdentifier

Example

```
extent1 := pom extentIdentifierFor: MicFwTutorialCustomer.
extent1 setQuery: [:customer | customer name lastName like:'M%'].

extent2 := pom extentIdentifierFor: MicFwTutorialCustomer.
extent2 setQuery: [:customer | customer addresses notEmpty].

combinedExtent := extent1 intersect: extent2.
```

There are some restrictions for combined queries:

- Only class queries that have the same result class can be combined
- The queries must not contain order clauses
- The queries must not use equal parameter names.

The underlying DBMS must support the combination feature.(It is important to set the correct database description class in the POM Configuration Editor.)

6.2.5. Query Parameters

A query can also contain parameters. Parameters must be defined inside the query using the *#parameterNamed:* method. The query value is supplied at evaluation time by sending *#parameterAt:put:* to the extent

identifier:

```
extent setQuery: [:address : queryEnv | | param |
param := queryEnv parameterNamed: 'person'.
address persons = param
].
extent parameterAt: 'person' put: somePerson.
extent load.
```

Within the query expression the parameter value is treated like a literal value.

Currently, parameters can only be used to directly represent a value. Sending messages to parameter values is not supported

6.2.6. Literals

Inside the query, any type of literal can be used. A literal is a value that is not a query variable or a path expression. Therefore, evaluation of pure Smalltalk expressions (without persistent semantics) within the query is possible:

```
[:person | person birthDate = Date today]
```

The expression "*Date today*" is evaluated as a Smalltalk expression because the receiver (class *Date*) is not a query variable or a path expression defined on a query variable. The person's birthdate is compared with the current date on your computer.

6.2.7. The Query Expression

The last expression in the query block is the query expression, which must evaluate to a Boolean. The query expression is made up of variables, path expressions, parameter variables, literal values and operators. Expressions can be nested to any level. The most simple expression is a comparison expression. Comparison expressions can be used in logical expressions.

The following operators are defined as having persistent semantics within the query condition:

Operator/ Message	Meaning	Example
and: or: & | = ~= <= < >= > + - * /	Exactly the same as for the Smalltalk version. Note that the arguments to and: and or: are written with parenthesis instead of brackets	[:user | user lastLoginDate < Date today and: (user loginCount ~= 0)]
= nil, ~= nil	For attributes: as SQL IS (NOT) NULL. For to-1 relationships : appropriate (NOT) EXISTS-clause	[:user | user name = nil] [:customer | customer ownerEmployee = nil]
like:, notLike:	As SQL (NOT) LIKE	[:user | user name like: 'Mic%']
between: and:	As SQL BETWEEN ... AND ...	[:user | user lastLoginDate between: (Date fromString: '01.12.96') and: (Date fromString: '15.12.96')]
in: aCollection, notIn: aCollection	Checks whether the left operand is (not) an element of the right operand	[:user | user id in: #(1 2 3)]
has:, hasNot:	Checks whether the left operand and the right operand have any/no element in common	[:user:env| | otherAddress | ... user addresses has: otherAddress]
isEmpty, notEmpty	Checks whether the operand has any/no elements	[:customer | customer addresses isEmpty]
orderBy:	For class queries inside of the query block	[:customer :env | env orderBy: #((name firstName) asc)].

Table 11: Operators for query conditions

The following operand types can appear in comparison expressions:

Type of LeftExpression or Right-Expression	Form
Value Literal	Arbitrary value from the Smalltalk image. May be defined inside the block (e.g. 'MyName'), outside the block or through a query parameter
PersistentObject Literal	Same as above, with the value class a subclass of *MicFwPersistentObject*
PersistentObject	- Local query variable - Path expression
PersistentObject collection	- Local query variable - Path expression

Table 12: Operand types for query conditions

A comparison expression must have a PersistentObject value on at least one side. Both sides of the comparison must evaluate to the same type. If a comparison is executed on PersistentObject values, the resulting SQL will compare all key columns that the corresponding class is mapped to. For example, in

```
[:address :queryEnv | | param |
param := queryEnv parameterNamed: 'person'.
address persons = param
].
```

a comparison is generated for all key columns the class Person is mapped to.

6.3. Projection Queries

With the aid of projection queries it is possible to use queries to obtain the attributes of various related classes. Unlike the previously described class queries, projections do not return completed persistent objects, but rather arrays with the specified attributes.

This is particularly useful if, for example, the attributes of several related objects are to be displayed together in a pick list, without having to fully instantiate the objects.

6.3.1. An Example

In the following example, the name of the customer and the name of the associated employee should be loaded for all customers whose ID's are between 20000 and 20005.

```
projection := pom newProjectionDescriptor.
projection setQuery: [:env |   | p1 p2 |
p1 := env newVariable.
p2 := env newVariable.
env
select: #((name lastName) (name firstName))
from: p1
in: MicFwTutorialCustomer;
select: #((name lastName) (name firstName))
from: p2
in: p1 ownerEmployee;
orderBy: #(id asc (name firstName) desc) of: p1; orderBy: #(id asc) of: p2.

(p1 id between: 20000 and: 20005) & (p2 id < 20010)].
^projection load.
```

The reply consists of an array for each customer/employee pair that has been found. The first array element contains the name of the customer; the second element contains the name of the employee.

Projections are described by a *Projection Descriptor* (Class ***MicFwPsProjectionDescriptor***), which is queried from the POM.

As you can see, there are two major differences between the above projection and a simple OQL:

- A select statement must be used, which is impossible in a simple OQL
- The result is not a collection of instances of a class but a collection of arrays containing attribute values.

6.3.2. The Query Block

The query block is similar to the normal class OQL, but with a few additional rules:

- The block variable (*:env in the example*) represents the environment variable that is required for defining variables. There is no result variable
- The classes and objects whose attributes should appear in the result are represented by *query variables* (*p1* and *p2* in the example)
- The query variables are defined together with the definition of the desired result attributes in the *select:* clause. The structure of the #select clause will be described in the next section.

6.3.3. The Select Clause

The result of a projection is defined by any number of select clauses. Each select clause has the following format:

```
env
select: #(<selector>, ..)
from: <queryVariable>
in: <aClass>
```

The clause consists of the following elements:

- The recipient of the *#select:from:in:* message is the query environment that is passed in the first parameter block.
- The *#select:* argument consists of a collection of symbols that represent the attributes of a persistent class. Aggregated attributes are specified using nested collections.
- The *#from:* argument specifies the query variable to which the clause relates.
- The *#in:* argument defines the assignment of the query variable to an extent. As in normal variable def-

initions, the extent can be defined by specifying a persistent class or a relationship

Constants can be added to the projection with the method *#selectConstants:*

Example

```
projection := pom newProjectionDescriptor.
projection setQuery: [:env |   | p|
p := env newVariable.
env
selectConstants: #( 111 'Text');
select: #((name lastName) (name firstName))
from: p
in: MicFwTutorialPerson.

(p id between: 0 and: 5000005)].
projection load.
```

6.3.4. Projection Operators

Projection descriptors can be combined with the following methods. The operators are equal to the SQL-operators and require, that the underlying DBMS supports the SQL features. Each operator returns a new combined query.

- union: aProjectionDescriptor
- intersect: aProjectionDescriptor
- except: aProjectionDescriptor

To define the sort order of a projection the method *orderByIndexes:* can be used. In comparison *to orderBy:of:* the method use numbers instead of selectors. The numbers refer to the indexes of columns in the answer set, for example:

```
projection orderByIndexes: #(1 asc 2 desc 3 asc 4 desc 5 asc).
```

6.4. Alternative Query Interfaces

The above descriptions covered the query block interface in which the query is represented by a Smalltalk block. The Mynd OQL implementation also provides 2 alternative query interfaces which better suit certain programming needs. These are:

- Array interface. In this interface, the query expression is specified as an array of symbols.
- Stream interface. This interface allows for programatical creation of queries.

Note: As opposed to the block interface, both these interfaces currently do not contain much error checking. It is therefore quite easy to create invalid expressions that will generate errors when converted to SQL.

6.4.1. Array Interface

With the Array interface, queries are expressed as Arrays of Symbols, as in:

```
MicFwTutorialCustomer where: #(name firstName = 'George' & id > 22)
```

The Array interface was mostly designed to provide compatibility with earlier PFW releases. Therefore, advanced features are not supported. These advanced features are:

- Variable definitions
- Query parameters
- Nested expressions

6.4.2. Stream Interface

The Stream interface was designed to allow programmatical construction of query expressions. It is modelled after Stream semantics with many messages prefixed "nextPut..". All the functionality of the block interface can be accessed through the stream interface. The stream interface is accessed by sending the *#queryStream* message to an extent identifier:

```
ext := pom extentIdentifierFor: MicFwTutorialCustomer.
ext queryStream
nextBeginQuery;
nextPutResult;
nextPutMessages: #(ownerEmployee name firstName);
nextPutComparison: #like:;
nextPutConstant: 'L%';
nextEndQuery.
ext load.
```

The code shown above is functionally equivalent to the following:

```
ext := pom extentIdentifierFor: MicFwTutorialCustomer.
ext setQuery: [:each |
each ownerEmployee name firstName like: 'L%'].
ext load.
```

The following example shows how to build the projection query given in [An Example: An Example] through the stream interface:

```
pro := pom newProjectionDescriptor.
pro queryStream
defineVariable: 'p1';
defineVariable: 'p2';
nextBeginQuery;
select: #((name lastName) (name firstName))
from: 'p1'
in: MicFwTutorialCustomer;
select: #((name lastName) (name firstName))
from: 'p2'
in: #(ownerEmployee)
on: 'p1';
orderBy: #(id asc (name firstName) desc) of: 'p1';
orderBy: #(id asc) of: 'p2';
nextPutVariable: 'p1';
nextPutMessages: #(id);
nextPutComparison: #between:;
nextPutConstant: (Array with: 20000 with: 20005);
```

```
nextPutAnd;
nextPutVariable: 'p2';
nextPutMessages: #(id);
nextPutComparison: #<;
nextPutConstant: 20010;
nextEndQuery.

projection := pro load.
```

The following list describes the public protocol of the class *MicFwQStreamParser* which is used to dynamically build queries as shown above:

- ***defineParameter: aName***. Define ***aName*** as a parameter variable.
- ***defineVariable: aName as: aMessageCollection on: otherVariable***. Define a variable under the name ***aName***. Declare it as representing the result of sending the messages in ***aMessageCollection*** to the (previously defined) variable ***otherVariable.***
- ***defineVariable: aName onResultAs: aMessageCollection***. Define a variable under the name ***aName***. Declare it as representing the result of sending the messages in ***aMessageCollection*** to the result variable.
- ***defineVariable: aName asExtentOf: aClass.*** Define a variable under the name ***aName***. Declare it as representing the extent of all persistent instances of ***aClass***.
- ***nextBeginQuery***. Begin the query.
- ***nextBeginSubExpr***. Begin a subexpression.
- ***nextEndQuery***. End the query. This must be sent before attempting to evaluate the query
- ***nextEndSubExpr***. End a subexpression
- ***nextPutAnd***. Append a logical AND to the query
- ***nextPutComparison: anOperator***. Append a comparison operator to the query. Valid parameters are #=, #~=, #>, #>=, #<, #<=, #like:.
- ***nextPutConstant: aValue***. Append a string or numeric constant to the query
- ***nextPutMessages: anArray***. Append a message sequence to the query.
- ***nextPutOr***. Append a logical OR to the query
- ***nextPutParameter: aName***. Append a reference to the parameter named ***aName*** to the query.
- ***nextPutResult***. Append a reference to the result variable to the query.
- ***nextPutVariable: aName***. Append a reference to the variable named ***aName*** to the query.
- ***select: aCollection from: aVariable in: aPersistentClass.*** Define a projection query extent variable and specify the attributes to be included in the projection answer set
- ***select: aCollection from: aVariable in: aPersistentClass on:***. Define a projection query relationship variable and specify the attributes to be included in the projection answer set

The methods for using subqueries with the stream interface are explained in section Subqueries for Stream Interface. and the usage of SQL-functions in a query stream is described in section Usage of SQL-functions in projection queries (stream-based OQL).

6.5. SQL-Functions

The OQL-operator named *function:* allows dealing with SQL-functions like DAY, MONTH or TRANSLATE in OQL-expressions.

The operator expects a kind of *MicFwRdbQueryFunction* as parameter. A separate subclass of *MicFwRdbQueryFunction* exists for each SQL-function that is currently supported.

Example

All customers, that have birthday today:

```
extent := pom extentIdentifierFor: MicFwTutorialCustomer.
extent setQuery: [:each |
((each function: (MicFwSQLFctDay attribute: #dateBirth)) =
Date today dayOfMonth)
&
((each function: (MicFwSQLFctMonth attribute: #dateBirth)) =
Date today monthIndex)
].
extent load.
```

Example

Query with a ForeignKey-Column compared with a given DB-Column:

```
extent := pom extentIdentifierFor: MicFwTutorialCustomer.
extent setQuery: [ :each |
(each function: (MicFwQFctColumn with: 'Customer.Employeeid'))
> 0
].
extent load.
```

Example

Query with explicit access to the subtype value:

```
extent := pom extentIdentifierFor: MicFwTutorialPerson.
extent setQuery: [ :each |
(each function: (MicFwQFctSubtype new)) like: '%Customer'
].
extent load.
```

6.5.1. Class hierarchy for supported SQL-functions:

Class	Description and Class-API
MicFwRdbQueryFunction	
MicFwQFctAttribute	Query an attribute
(with: anAttributeSymbol)	MicFwQFctColumn
Query a database column (aColumnString is case sensitive!)	*(with:* aColumnString)
MicFwQFctSubtype	Query the subtype value. (Create a new instance with `new` sent to the function class.)
MicFwSQLFunction	Abstract class, that implements APIs #attribute:, #column: and #function:. #function expects a QueryFunction, so that nested SQL-functions can be build. *(attribute:* anAttributeSymbol) *(column:* aColumnString) *(function:* aFunction)
MicFwSQLFctDay	SQL-Function `DAY(column)`
MicFwSQLFctMonth	SQL-Function `MONTH(column)`
MicFwSQLFctYear	SQL-Function `YEAR(column)`
MicFwSQLFctHour	SQL-Function `HOUR(column)`

Class	Description and Class-API
MicFwSQLFctMinute	SQL-Function `MINUTE(column)`
MicFwSQLFctSecond	SQL-Function `SECOND(column)`
MicFwSQLFctSubstr	SQL-Function `SUBSTR(column, startPos, string-Length)` *(attribute:* anAttribute *start:* startPos *length:* string-Length) *(column:* aColumn *start:* startPos *length:* stringLength) *(function:* aFunction *start:* startPos *length:* stringLength)
MicFwSQLFctTranslate	SQL-Function `TRANSLATE(column, fromString, toString)` *(attribute:* anAttribute *from:* fromString *to:* toString) *(column:* aColumn *from:* fromString *to:* toString) *(function:* aFunction *from:* fromString *to:* toString)
MicFwSQLFctUpper	SQL-Function `UPPER(column)`
MicFwSQLFctLower	SQL-Function `LOWER(column)`
MicFwSQLFctLength	SQL-Function `LENGTH(column)`
MicFwSQLFctAbs	SQL-Function `ABS(column)`
MicFwSQLFctAvg	SQL-Function `AVG(column)`
MicFwSQLFctMax	SQL-Function `MAX(column)`
MicFwSQLFctMin	SQL-Function `MIN(column)`
MicFwSQLFctSqrt	SQL-Function `SQRT(column)`
MicFwSQLFctSum	SQL-Function `SUM(column)`

Table 13: Class hierarchy for supported SQL functions.

6.5.2. How to build additional query functions

To use own/additional SQL-functions from inside OQL-queries, you can subclass the class *MicFwSQL-Function*.

You must pay attention to the following:

Overwrite the instance-method *#getSqlFunctionWith: aColumnString*, that return a SQL-function-String. *aColumnString* is a column name with alias name. The alias name generated by the framework is a parameter for the method.

Example

```
getSqlFunctionWith: aColumnString
"Public - answer a String with the SQL-Function including the
 column @aColumnString.
In this case the SQL-standard-function ABS"

^'ABS(', (self getSqlFor: aColumnString), ')'
```

The method *#getSqlFor*: returns the SQL-String of nested functions.

Implement the instance-method *#type*, to return either a Smalltalk datatype like Integer, String or Date or nil, if the datatype will not be changed by the SQL-function.

Optionally you can overwrite the methods #converterClass, #precision and #scale.

6.5.3. Usage of SQL-function with Stream Interface

The operator *#nextPutFunction*: allows to use the function-feature with the streambased OQL interface.

Example

```
extent := pom extentIdentifierFor: MicFwTutorialCustomer.
extent queryStream
nextBeginQuery;
nextPutResult;
```

```
nextPutMessages: #(ownerEmployee);
nextPutFunction: (MicFwSQLFctDay attribute: #dateBirth);
nextPutComparison: #=;
nextPutConstant: Date today dayOfMonth;
nextPutAnd;
nextPutResult;
nextPutMessages: #(ownerEmployee);
nextPutFunction: (MicFwSQLFctMonth attribute: #dateBirth);
nextPutComparison: #=;
nextPutConstant: Date today monthIndex;
nextEndQuery.
```

6.5.4. Usage of SQL-function with projection queries (block-based OQL)

It is possible to select attributes, that have been modified by a SQL-function with a projection query. Two operators are provided to support this:

- *#selectFunction: aFunction from: aVariable*
- *#selectFunction: aFunction from: aViriable in: aPersistentClass.*

Example

```
projection := pom newProjectionDescriptor.
projection setQuery: [:env| |p|
p := env newVariable.
env
selectFunction:(MicFwSQLFctLength
attribute:#(name lastName))
 from: p
in: MicFwTutorialCustomer;
select: #((name lastName)) from: p;
selectFunction: (MicFwQFctSubtype new) from: p.

(p name firstName like: '%')
& ((p function: (MicFwQFctSubtype new)) like: 'M%')
].
```

6.5.5. Usage of SQL-functions in projection queries (stream-based OQL)

The following three operators allows the usage of SQL-functions for projection queries with the stream-based OQL interface.

- #selectFunction: aFunction from: aVariable
- #selectFunction: aFunction from: aVariable in: aPersistentClass
- #selectFunction: aFunction from: aVariable in: aPersistentClass on: anotherVariable

Example

```
projection := pom newProjectionDescriptor.
projection queryStream
defineVariable: 'p1';
defineVariable: 'p2';
nextBeginQuery;
selectFunction: (MicFwSQLFctLength
attribute: #(name lastName))
from: 'p1'
in: MicFwTutorialCustomer;
select: #((name lastName) ) from: 'p1';
selectFunction: (MicFwQFctColumn with:'Customer.Employeeid')
from: 'p1';
selectFunction: (MicFwSQLFctLength
attribute: #(name lastName))
from: 'p2'
in: #(ownerEmployee)
on: 'p1';
select: #((name lastName) ) from: 'p2';
nextPutVariable: 'p1';
```

```
nextPutMessages: #(id);
nextPutComparison: #between:;
nextPutConstant: (Array with: 5000000 with: 5000010);
nextPutAnd;
nextPutVariable: 'p2';
nextPutMessages: #(id);
nextPutComparison: #<;
nextPutConstant: 1000008;
nextEndQuery.
```

6.6. Subqueries

6.6.1. Subqueries for Query Block Interface

In some cases nested query blocks are needed. For example subqueries must be used, if there are and-linked conditions for objects of a relationship. In the following example all customers should be loaded, who have at least one address in the city 'Hennef' in a street which name begins with 'Re'.

The attempt to define the query without subqueries (see below) returns a wrong result, because the created SQL statement loads also all customers, who have one address in the right city with the wrong street and another address in the wrong city with the right street.

Attempt **without** Subquery:

```
pom getInstancesOf: MicFwTutorialCustomer
where:[:customer :queryEnv |
| addr |

"define and declare query variable"
addr := queryEnv newVariable.
addr in: customer addresses.

(addr cityName = 'Hennef')
and:
(addr streetName like: 'Re%')
].
```

To get the wanted result, the conditions for the address must be defined in one subquery. A subquery can be defined by sending the message *newSubQueryOn:as:* to the query environment (see example below).

Object Query **with** Subquery:

```
pom getInstancesOf: MicFwTutorialCustomer
where:[:customer :queryEnv | | addr |
addr :=
queryEnv newSubqueryOn: customer addresses as:
[:a |
(a cityName = 'Hennef')
and:
(a streetName like: 'Re%')
].
addr notEmpty].
```

6.6.2. Subqueries for Stream Interface

Use one of the following methods to define a variable for a subquery with the stream interface.

- defineVariable: aName asSubqueryOnResult: aMessageCollection
- Define a variable under the Name *aName*. Declare it as representing the result of sending the messages in *aMessageCollection* to the result variable.
- defineVariable: aName asSubqueryOn: otherVarName with: aMessageCollection
- Define a variable under the Name *aName*. Declare it as representing the result of sending the messages in *aMessageCollection* to the (previously defined) variable *otherVarName*
- defineVariable: aName asSubqueryOn: aClass
- Define a variable under the Name *aName*. Declare it as representing the extent of all persistent instances of *aClass*.

The method `queryStreamOn:aName` returns the query stream for the variable named `aName`. The *subquery* can be defined between the calling of the methods `nextBeginQuery` and `nextEndQuery` for this stream.

The method `nextPutResult` puts the result variable of the methods receiver stream to the stream.

Example

load all customers, who have at least one address in 'Hennef' where the street name begins with: 'Re'

```
|extent mainQS subQS |

extent := pom extentIdentifierFor: MicFwTutorialCustomer.
```

```
mainQS := extent queryStream.
mainQS
nextBeginQuery;
defineVariable: 'addr' asSubqueryOnResult: #( addresses ).

subQS := mainQS queryStreamOn: 'addr'.
subQS
nextBeginQuery;
nextPutResult;
nextPutMessages: #(cityName);
nextPutComparison: #=;
nextPutConstant: 'Hennef';
nextPutAnd;
nextPutResult;
nextPutMessages: #(streetName);
nextPutComparison: #like: ;
nextPutConstant: 'Re%';
nextEndQuery.

mainQS
nextPutVariable: 'addr';
nextPutNotEmpty;
nextEndQuery.

extent load
```

6.7. Error messages

The following error classes are signaled when executing queries:

Error class	Type	Description
MicFwQueryError	General	Superclass of all query error classes. Can be used as a catch-all.
MicFwQParseError	Parsing	Signals an error during the parsing of the query block.
MicFwQBuildError	SQL conversion	Signals an error during conversion of the query in SQL
MicFwQParameterError	Parameter-related	Signals a parameter-related error, e.g. too few values have been set.

Table 14: List of error classes for queries

A more detailed description of the individual errors can be found in the exception object message text.

7

Optimistic Concurrency

7.1. Introduction

7.1.1. What does optimistic concurrency mean ?

Pessimistic concurrency handles multi-user situations in an application either by explicit object locking (e.g. in a lock table), database locking (table or row locking) or implicit locking when using a high isolation level.

Optimistic concurrency is the opposite of pessimistic concurrency which uses locking to prevent conflicts between users. Whereas pessimistic concurrency means that "user A" has to wait until "user B" has finished modifying data when they want to access the same data rows, optimistic concurrency does not require this kind of locking and thus allows all users to make changes at will. Because this strategy is called "optimistic", it is assumed that conflicts do not occur very frequently.

The use of such a low isolation level means that the application must be informed when a conflict has occurred. This means that conflicts are not disabled by locking and can occur (not very often, hopefully). When they do occur, they must be detected and the application must react accordingly.

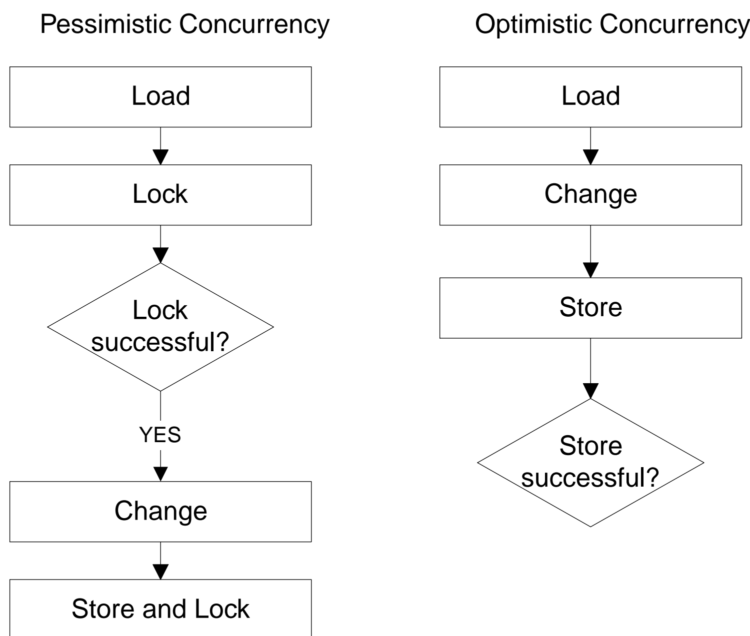


Figure 41: Pessimistic vs. Optimistic Concurrency

7.1.2. What are concurrency conflicts?

This chapter describes four kinds of concurrency conflicts from an application point of view. The following chapters refer to these types of conflict and explain how to deal with them in an application.

In general, a conflict is a situation where "user A" wants to change something in the database (inserting, updating or deleting) but another user (let's say "user B") has already made a change which affects the changes made by "user A".

Here are some examples of what can happen:

7.1.2.1. Example: Duplicate primary key

"User A" wants to insert a row of data into a table, but this table already contains a row with the same pri-

mary keys (e.g. inserted by "user B" shortly beforehand).

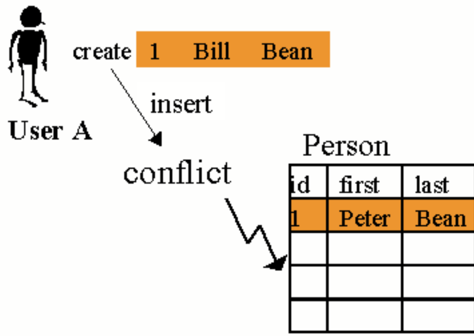


Figure 42: Duplicate primary key conflict

7.1.2.2. Example: Differences when re-reading objects:

"User A" has some persistent objects in his Smalltalk image. Meanwhile "user B" changes some of these objects in the database. Thus "user A" gets changed instances when re-reading them from the database.

7.1.2.3. Example: Deleted objects

"User A" tries to commit his changes to the database, but "user B" has deleted the underlying rows in the database. "User A" must be informed about this conflict.

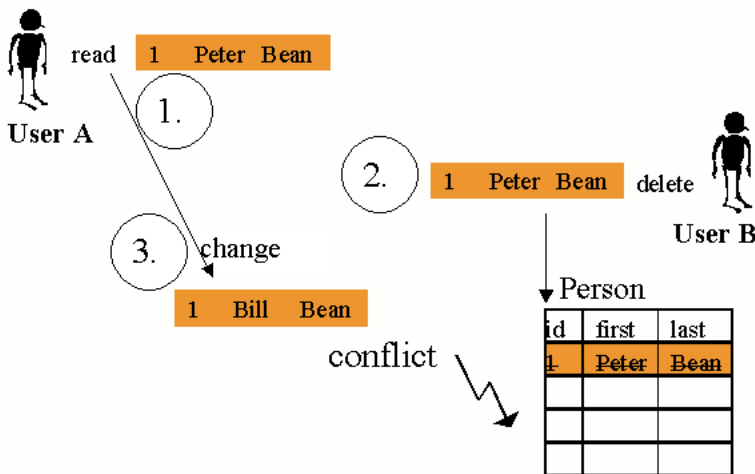


Figure 43: Deleted object causing an update conflict

7.1.2.4. Example: Concurrent changes

"User B" has a persistent object in his Smalltalk image and changes it. "User A" also changes the same object and commits his changes to the database. "User B" must be informed that the database contents have changed when he tries to commit his changes, which are no longer derived from the previous state of the database.

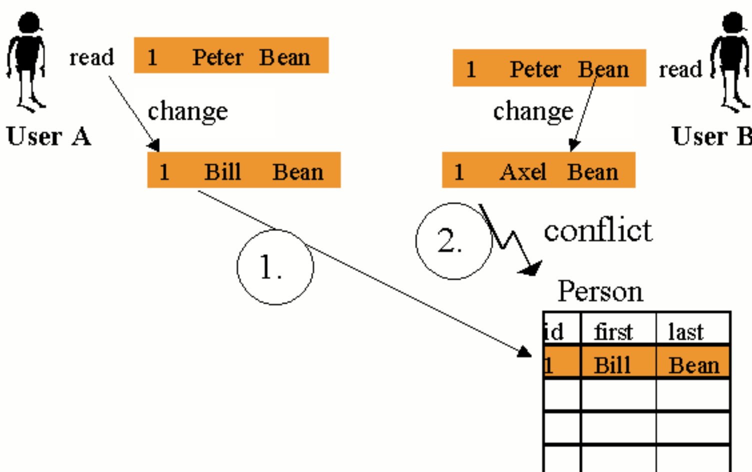


Figure 44: Concurrent change causing an update conflict

7.1.2.5. Example: Deletion conflict after concurrent change:

"User A" has a persistent object in his Smalltalk image. "User B" changes the same object and commits his changes to the database. "User A" then tries to delete the object, but the object in his image is no longer derived from the current state in the database. So "User A" must be informed that his deletion is invalid.

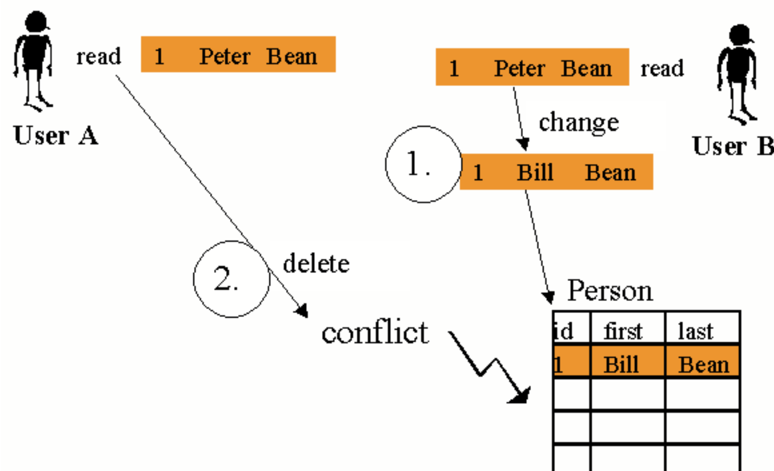


Figure 45: Concurrent change causing a delete conflict

7.1.3. Dealing with conflicts

This chapter deals with how an application should react when a conflict occurs. We will refer to the conflicts described in the previous chapter and give a general explanation of the possible actions that the application might take.

7.1.3.1. Example: *Duplicate primary keys*

No special requirements exist for handling this conflict in systems using optimistic concurrency. In any event, the user gets an "integrity violation" from the database when trying to insert duplicate primary keys or when updating them in such a way that they are no longer unique. The user or the application may want to

- skip the operation that caused the conflict and continue with the next operation.
- repair the conflicting values (e.g. choose a valid primary key value) and repeat the operation.
- stop committing the context, possibly aborting the whole context.

7.1.3.2. Example: Differences when re-reading objects:

This kind of conflict can occur when the database connection runs in a low isolation level or when instances read from the database are used for longer than the duration of a database transaction.

The user may ignore this conflict and just rely on the objects which are cached in the image. In this case, he is not interested in changes made by other users while using persistent instances for read access.

The user wants to be informed when objects that are read from the database differ from objects which are already cached in the image. The user may decide to:

- Use the old cached object (ignoring the changes from the database)
- Use the current object from the database (which is not identical to the cached one)
- Copy the contents of the current object into the cached object. (Ignoring pending transaction context changes)
- Refresh the contents of the current object with the contents of the cached object. (This also means that changes which may be pending in open transaction contexts are checked and cleared up, if required by the user).

7.1.3.3. Example: Deleted objects

When a user is informed of this kind of conflict, he can decide to:

- make the persistent object transient, so that it is already marked for deletion.
- re-insert this persistent object back into the database.

7.1.3.4. Example: Concurrent changes:

This kind of conflict requires closer examination, because the application's requirements determine how this conflict should be detected, and under which conditions an exception should be generated.

The following options are conceivable, where an object (e.g. Customer(100)) is to be updated in the data-

base:

The exception will be triggered if another user changes anything to do with Customer(100). This means that all the attributes of Customer(100) must still be the same as those read from the database when the object was loaded. This means that the conflict is signaled at instance level.

Other users may have changed some attributes of Customer(100) in the meantime. The conflict will only be raised when the other user has changed one of the attributes to be updated. This means that the conflict is raised at instance variable level.

The exception will be triggered if another user has changed attributes in the same row that is going to be updated. This means that the conflict is detected at table level or database row level. This granularity is finer than "Instance comparison" but not as detailed as "Change comparison". This option can detect conflicts in a performant way.

7.1.3.5. Example: Deletion conflict after concurrent change:

If this situation should be regarded as a conflict or not, is dependent on the application's requirements.

It might not be a conflict to delete an object that another user has just changed, when deletion has a higher priority than change for an application. If the situation is a conflict or not might also depend on what attributes have been changed. The deletion may not only fail because of other users changes, but also when another user already has deleted the object or has changed the object's primary key attributes.

The Mynd Persistence Framework" supports the use of any column that is directly mapped to instance variables or aggregation instance variables as "Optimistic column". This provides many flexible configurations. We will describe the required preparations using the STOPF mapping tool in the following chapters.

Finally, when a concurrent change conflict is signaled, the user may decide to:

- skip the update of this instance.
- update this instance anyway and overwrite the changes made by other users.

7.1.4. Configuring classes for optimistic concurrency

When using the "Mynd Persistence Framework" you have a couple of options for configuring your persistence manager to deal with the special optimistic concurrency requirements in the application.

For each class mapped in a persistence manager (POM), you can set the appropriate optimistic strategy attribute for those columns you want to use for update-conflict-detection. Refer to 'Optimistic Strategy' (page 89) and 'Setting column mapping attributes' (page 203) for concepts and tool support.

You may have a timestamp column mapped with an optimistic strategy in each database table to which your classes are mapped. Each time an instance is inserted or updated, this "optimistic" timestamp column has to be set to the current timestamp. The conflict detection uses this timestamp value in all UPDATE statements as follows:

The "normal" pessimistic UPDATE-SQL statement could look like this (ID is the only primary key column of table PERSON):

```
UPDATE "DBA".PERSON SET FIRSTNAME='Peter'
WHERE ID=100
```

When using optimistic concurrency with a timestamp column (e.g. column name = TIME_STAMP), the Framework generates:

```
UPDATE "DBA".PERSON SET FIRSTNAME='Peter'
WHERE ID=100 AND TIME_STAMP='1996-12-06-11.19.49.000000'
```

After executing the statement, the Framework uses the rows-affected result of the statement to check whether the specified row (with id and timestamp) could be updated. The result for rows which are normally affected is 1 and no conflict occurs. If the result is 0, the Framework raises an exception because another user has probably changed something in this row (because the timestamp is different) or deleted it.

7.1.4.1. Value strategies of an optimistic timestamp column

When using optimistic columns (e.g. timestamps) for the update-conflict-detection of a class, it is important that the column gets updated with the new value (e.g. the current timestamp) every time the row is hit by an SQL statement. You can use the value and reload strategy of the column to configure the expected behavior.

Here are three examples when working with optimistic timestamp columns: The value can be provided by:

- a trigger defined on the database
- CURRENT TIMESTAMP in the generated SQL statements
- an appropriate value message

It is also important that the new timestamp value is taken from a central point (e.g. the database server) in

order to ensure that the same timestamp is not used twice.

7.1.4.1.1. Example: A trigger defined on the database

This means that a database trigger automatically updates the timestamp column before or after every INSERT and UPDATE statement on this table. The Framework should rely on this trigger and not modify the value of the timestamp itself. The trigger has to be defined by the user (e.g. the database administrator) and could look like this:

This example trigger updates the TIME_STAMP column in table PERSON for every INSERT or UPDATE:

DB2 Trigger Syntax

```
CREATE TRIGGER MyUpdateTrigger
NO CASCADE BEFORE UPDATE ON "DBA".PERSON
REFERENCING NEW AS new_row
FOR EACH ROW MODE DB2SQL
SET new_row.TIME_STAMP = CURRENT TIMESTAMP

CREATE TRIGGER MyInsertTrigger
NO CASCADE BEFORE INSERT ON "DBA".PERSON
REFERENCING NEW AS new_row
FOR EACH ROW MODE DB2SQL
SET new_row.TIME_STAMP = CURRENT TIMESTAMP
```

Sybase SQL Anywhere Syntax

```
CREATE TRIGGER MyUpdateTrigger AFTER UPDATE ON "DBA".PERSON
REFERENCING NEW AS new_row OLD AS old_row
FOR EACH ROW
   BEGIN
      UPDATE "DBA".PERSON
      SET TIME_STAMP = CURRENT TIMESTAMP
      WHERE ID= new_row.ID
   END

CREATE TRIGGER MyInsertTrigger AFTER INSERT ON "DBA".PERSON
REFERENCING NEW AS new_row
FOR EACH ROW
   BEGIN
      UPDATE "DBA".PERSON
      SET TIME_STAMP = CURRENT TIMESTAMP
         WHERE ID = new_row.ID
   END
```

Because of the database trigger doing all the value modifications on this column for you, the attribute configuration could look like this:

Value Strategy (for insert and update): Ignore

Reload Strategy (for insert and update): Reload after statement to get the new value from the database.

Pro: The trigger works independently of the Framework and thus even updates the timestamp if other applications modify the row, or direct SQL statements are performed in order to modify the row.

7.1.4.1.2. Example: CURRENT TIMESTAMP in the generated SQL statements

When no trigger is defined in the database, you can let the Framework generate SQL statements with an SQL expression included, that writes the current timestamp to the database each time the table row is being updated.

Attribute configuration:

Value Strategy: SQL Constant, e.g. CURRENT TIMESTAMP

Reload Strategy: Reload after statement to get the new value from the database.

When using this solution, a statement to update the PERSON table would look like this:

```
UPDATE "DBA".PERSON
SET FIRSTNAME='Peter',
TIME_STAMP=CURRENT TIMESTAMP
WHERE ID = 100 AND
TIME_STAMP='1996-12-06-11.19.49.000000'
```

Pro: This solution is similar to the former but you do not need trigger definitions in the database.

7.1.4.1.3. Example: an appropriate value message

When using triggers or the SQL constant, the new value which is actually written to the row is unknown to the application program after the statement has been performed.

In order to obtain the new timestamp value, the column has to be explicitly re-read afterwards. For this reason you needed a reload strategy to have the current value in the persistent object again. This additional statement generates an overhead that can be avoided when using a value strategy with a value message. The update statement for would look like this:

```
UPDATE "DBA".PERSON
SET FIRSTNAME='Peter',
TIME_STAMP='1996-12-06-11.22.05.000000'
WHERE ID = 100 AND
TIME_STAMP='1996-12-06-11.19.49.000000'
```

This means that the concrete new value is set by the same update statement and there is no need to re-read the new value subsequently. The new timestamp value can be taken from the database server, but this can be done by the Framework only once when committing a persistence context. The statement to prefetch the new timestamp from the database server could be (Sybase SQL Anywhere syntax):

```
SELECT CURRENT TIMESTAMP
```

Note: DB2 requires a FROM clause in the statement. You can either define a stored procedure or user function under DB2 which returns a new timestamp or your can "abuse" any table that has at least one row of data in it, e.g. SELECT CURRENT TIMESTAMP FROM SYSCAT.TABLES (or with the following SQL-statement: VALUES (CURRENT TIMESTAMP)). It may also be possible to obtain the new timestamp from somewhere else other than the database server. Just make sure that the timestamp comes from a central machine in order to guarantee uniqueness.

Attribute configuration:

Value Strategy: Value Message, e.g. timestamp

Reload Strategy: None

The value message #timestamp will be sent to the current commit handler of the POM. The commit handler performs an SQL statement named #getServerTimestamp from the POM's registry and fetches the result of the first row of the answer set. The commit handler performs this statement only once for a transaction commit. So you will get only one additional statement as an overhead in contrast to the many SQL statements with reload strategy solutions.

Make sure that your POM has a direct statement to fetch the server timestamp registered in the POM's statement registry. The statement must be registered with registry key #getServerTimestamp.

Check whether your POM has registered a statement called #getServerTimestamp by evaluating the following:

```
((MicFwPersistenceManager named: 'MyPOM')
hasRegistryForKey: #getServerTimestamp)
ifTrue: [ MicSystem messageBox
message: 'OK. Registry key #getServerTimestamp found.' ]
ifFalse: [ MicSystem messageBox
message: 'No registry key named #getServerTimestamp found.' ].
```

Register the prefetch statement. You can also do this with the statement registration tool within STOPF: (see ‘Statement Registration Tool’ (page 189))

```
| stmt pom |
pom := MicFwPersistenceManager named: 'MyPom'.
stmt := pom sqlDescriptorFor: 'SELECT CURRENT TIMESTAMP'.
pom registerStatement: stmt as: #getServerTimestamp.
```

Normally a commit of a persistence context is more than one or two update or insert statements. The Framework prefetches the new timestamp only when it is first needed and only once in a commit phase. The more objects that have to be updated or inserted, the more performance benefits you have.

You can implement your own value messages in the commit handler you are using, e.g. to call a stored procedure etc. Then you do not need a registered #getServerTimestamp statement.

Pro: This strategy gives the best performance because the statement overhead is reduced to a minimum.

These examples showed you some possible configurations when using timestamp columns. Analogous configurations can be chosen or implemented to use any other column type for optimistic concurrency.

7.1.5. Handling update conflicts

This chapter describes how the Framework supports the handling of update conflicts: "deleted objects"

and "concurrent changes" mentioned in chapter What are concurrency conflicts? What are concurrency conflicts?.

7.1.5.1. Notifications/Exceptions raised inside #commitTransaction

The kind of notification triggered in response to an exception where an object to be updated is deleted or modified by another user depends on the strategy used to detect a conflict and the commit handler performing the context commit.

In any event, we must assume that conflicts occur quite rarely, otherwise an application would not use optimistic concurrency.

All optimistic concurrency-specific conflicts which might occur are triggered while the ***context>>#commitTransaction*** method is running (except for optionCompare changed conflicts which are also triggered for implicit load operations. See previous chapter). This chapter deals with conflicts raised while the #commitTransaction method is running.

If a conflict occurs, the Persistence Framework always raises the type of exception which most accurately describes the conflict, depending on the information the Framework has about the instance producing the conflict.

Resumable notifications

- ***MicFwPersistenceConflict*** . Abstract / general notification for optimistic conflicts
- ***MicFwPsCompareConflict.*** An instance is somehow different from its persistent state (or the instance just loaded from the DB).
- ***MicFwPsAspectCompareConflict*** . Like MicFwPsCompareConflict, the aspect (instance variable) that produced the conflict is also known
- ***MicFwPsDeletedConflict*** . An instance could no longer be found on the database and has probably been deleted.
- ***MicFwPsUpdateConflict.*** Updating the object on the database failed because optimistic values are invalid. This is the exception when working with optimistic columns.
- Non resumable errors
- ***MicFwPersistenceError.*** Abstract /general class for conflicts that are linked with a persistent object.
- ***MicFwPsUpdateError.*** Non-resumable sibling of MicFwPsUpdateConflict. Raised when MicFwPsUpdateConflict was not handled.
- ***MicFwRdbIntegrityError.*** Non-resumable sibling of MicFwRdbIntegrityViolation. Raised when MicFwRdbIntegrityViolation was not handled.

Integrity violations are independent of optimistic concurrency.

7.1.5.2. MicFwPersistenceConflict - abstract class

Method ***#instance*** returns the persistent object that caused the conflict.

Method ***#preferPersistence***: tells the Framework whether to respect / use (=true) the persistent state or not (=false).

The defaultAction opens a messageBox and lets the user decide how to continue.

7.1.5.3. MicFwPsCompareConflict

This exception is triggered when one of the methods ***MicFwPersistentObject>>persistenceCompare: persistentObject***, ***MicFwPersistenceManagerRdb>>persistenceCompareTo: persistentObject*** finds differences between an object and its representation on the database or the other object.

7.1.5.4. MicFwPsAspectCompareConflict - special kind of MicFwPsCompareConflict

#aspect

A symbol (instVar-Accessor) for the instance variable which has been changed in the DB **or** an array with two symbols (when notifying a change for an aggregated object). Two symbols: 1. AggregateRelationship-Selector 2. Aspect-Selector in the aggregated object, e.g. #(name firstName).

#newValue

the new value from the DB for this aspect.

This exception is triggered when using the method ***MicFwPersistentObject>>refresh***. The conflict occurs when a persistence object has an aspect that is modified in the DB **and** a version is pending in a transaction context for this aspect.

7.1.5.5. MicFwPsDeletedConflict

This exception is triggered when using one of the persistenceCompare or refresh methods and the underlying persistent object has been deleted from the database.

This exception also is triggered when a delete operation with optimistic concurrency fails. In this situation

the exception does not automatically indicate, that the underlying object has been deleted. The deletion could also have no affected rows, because of a concurrent primary key or optimistic column change.

7.1.5.6. MicFwPsUpdateConflict

- ***#updateBlock*** returns a zero argument block which updates the object data anyway when sending the #value message.
- ***#insertBlock*** returns a zero argument block which inserts the object again when sending the #value message.
- ***#transientBlock*** returns a zero argument block which makes the conflict's instance transient when sending the #value message.
- ***#preferPersistence*** has no meaning here.

The defaultAction triggers the non-resumable ***MicFwPsUpdateError.***

This kind of exception is triggered when a column uses an optimistic strategy and commit handler ***MicF-wRdbCmtHdlrOptimistic*** **or one of its subclasses** is used.

The Framework notifies the application that an update statement failed. The only thing that is known at this moment is that the value(s) have changed in the database. This could mean that the persistent object was modified by another user, but it is also possible that the persistent object may have been deleted by another user. When the application handles a ***MicFwPsUpdateConflict***, it is left to the application to detect whether or not the object has been deleted (this is not done by the Framework automatically, because it would result in an extra SQL statement for detecting the real cause of the conflict).

The application could do the following:

```
[ context commitTransaction ]
  on: MicFwPsUpdateConflict
  do: [ :conflict |
    self log: conflict messageText.
    "** detect whether the object was deleted or modified **"
    conflict instance persistenceIdentifier existsInDB
      ifTrue: [
        self log: conflict instance asString, ' has been modified.'.
        (MicSystem messageBox confirm: 'Overwrite the modifications?')
          ifTrue: [
            "** update instance and overwrite changes of other users **"
            conflict updateBlock value
          ]
      ]
      ifFalse: [
        self log: conflict instance asString, ' has been deleted.'.
        (MicSystem messageBox confirm: 'Insert again? (Or make transient?)')
          ifTrue: [
            "** reinsert object although been deleted by another user **"
            conflict insertBlock value.
          ]
          ifFalse: [
            conflict transientBlock value.
          ]
      ]
  ].
```

7.1.6. Choosing the POM's commit handler

Use the POM Configuration Editor (described in ‘POM Configuration Editor’ (page 187)) to modify the configuration of your persistence manager.

Typical persistence manager configurations:

7.1.6.1. Pessimistic POM:

No column mapping has an optimistic strategy attribute.

Configuration:

- Option compare: OFF or ON
- Commit Handler: ***MicFwRdbCommitHandler*** [no optimistic strategy supported]

7.1.6.2. Optimistic POM:

Some column mappings have an optimistic strategy attribute.

Configuration:

- Option compare: ON [recommended]
- Commit Handler: ***MicFwRdbCmtHdlrOptimistic***

7.1.7. API methods

This chapter describes API methods for persistent objects. These methods are not intended for optimistic concurrency applications only. There might also be other situations where these methods could be used.

7.1.7.1. MicFwPersistentObject>>resetRelationships

Send #resetLoaded to all persistent relationships of the receiver. This causes the relationships to be resolved from the persistence medium the next time they try to get their target(s).

7.1.7.2. MicFwPersistentObject>>persistenceCompareTo: loadedObject

Compare the receiver to <loadedObject>, which was just loaded from the persistence medium and found to have the same identity as the receiver. If the objects are different, signal a ***MicFwPsCompareConflict***.

7.1.7.3. MicFwPersistentObject>>persistenceCopyFrom: loadedObject

Copy all persistence-relevant data from <loadedObject>, which was just loaded from the persistence medium and found to have the same identity as the receiver, but different persistent data. Copy all relevant data into the receiver. The default implementation lets the persistence manager handle the job.

7.1.7.4. MicFwPersistentObject>>refresh

Refresh the receiver's persistent instance variables by loading and comparing the persistence medium values. ***MicFwPsDeletedConflict*** will be triggered if the instance could no longer be found in the database.

MicFwPsAspectCompareConflict will be triggered for any aspect that has a version in a transaction context and is modified in the database (when the application sets the preferPersistence: state to <true> the aspect will be refreshed and the versions in the contexts will be removed).

Copy the current values from loadedObject into the receiver.

7.1.7.5. MicFwPersistentObject>>refreshWith: loadedObject

Refresh the receiver to <loadedObject>, which was just loaded from the persistence medium and found to have the same identity as the receiver. If the objects are different, do whatever is appropriate in that situation (e.g. copy loaded data, notify the user). The default implementation lets the persistence manager do the comparison. Signal a ***MicFwPsAspectCompareConflict*** if it fails.

Copy the current values from loadedObject into the receiver.

7.1.7.6. MicFwPersistentObject>>invalidateRelationships

Send #invalidate to all persistent relationships for the receiver. This causes the relationships to be resolved from the persistence medium the next time they try to get their target(s). All the versions for these relationships are removed from the transaction versioning mechanism.

7.1.7.7. MicFwPs1Relationship>>invalidate, MicFwPsNRelationship>>invalidate, *MicFwPsPrim1Relationship>>invalidate*

Set the state of the receiver to be unresolved from the persistence medium. This causes the receiver to load the persistent objects from the persistence medium the next time this is necessary.

Remove all versions that include invalid object versions and make all changes in affected relationships.

7.1.7.8. MicFwPersistenceManagerRdb>>persistenceCompare: oneObject

Compare the values of @*oneObject* with the persistent representation. Signal a ***MicFwPsCompareConflict*** or a ***MicFwPsDeletedConflict*** for any changes. Otherwise signal nothing and answer self.

7.1.7.9. MicFwPersistenceManagerRdb>>persistenceCompare: oneObject to: anotherObject

Compare the value equivalence of two objects. Answer true if equal, false otherwise. This implementation uses low level read operations on the objects (i.e. #instVarNamed:) and thus bypasses the transaction mechanism. The compared state is the one that existed when the objects were loaded. Answers true when the objects are equal, otherwise false.

7.1.7.10. MicFwPersistenceManagerRdb>>persistenceCopyTo: oneObject from: anotherObject

Copy the persistent data from <anotherObject> to <oneObject>. The copy is done for all non-key variables using low level accessors (#instVarNamed:) and thus affects the basic state of the object to which it is cop-

ied. Any changes within the current transaction are still retained and will be committed if not discarded. This method only needs to be re-implemented if any persistent object manipulated through the receiver retains the default implementation of ***MicFwPersistentObject>>persistenceCopyFrom:***.

7.1.7.11. MicFwPersistenceManagerRdb>>refresh: persistentObject

Refresh the persistent instance variables of @*persistentObject* by loading the associated data from persistence and comparing the values. Signal adequate exceptions if there are inconsistencies. The following inconsistencies are signaled:

a) concurrent changes in persistence and current transaction context

b) object deleted from persistence.

7.1.7.12. MicFwPersistenceManagerRdb>>refresh: persistentObject with: otherObject

Refresh the persistent instance variables of @*persistentObject* by comparing with the values of @*otherObject*.

Note: @*otherObject* and @*persistentObject* should be instances of the same class (e.g. reloading the same object). Signal appropriate exceptions if inconsistencies are detected.

7.1.7.13. MicFwPersistenceManagerRdb>>optionCompare: aBoolean

Set whether any loaded object will be compared to the cached representation when instances are read (explicitly or on relationship traversal).

8

Embedded SQL

8.1. Introduction

8.1.1. What is Embedded SQL?

Many RDBMS provide several alternative ways of accessing DBMS functionality. The most common ones are *call level interfaces* (CLI) and *embedded SQL*. With call level interfaces, calls to a DBMS-dependent API are coded into the program. SQL statements are passed in string form and processed by the DBMS at runtime. With embedded SQL, SQL statements are written in a special syntax which is embedded within the host language and marked up by delimiters. The statements are picked up by a preprocessor at compile time and passed to the DBMS, where they are converted to an optimized form. Here is an example of an SQL statement embedded in a COBOL program:

```
move 22 to counter.
EXEC SQL
SELECT * FROM TABLE_X WHERE NUM = :counter
END EXEC
```

The SQL statement is found between the delimiters *EXEC SQL* and *END EXEC.* The precompiler passes the SQL statements to the DBMS and replaces them with parameterized function calls in the host program. The SQL statements are put together into a unit called a package which is then bound to the database (either at precompile time or later). During the bind process the statements are run through the DBMS query optimizer and stored inside the DBMS.

8.1.2. Dynamic and Static SQL

Two other terms that need to be mentioned in this context are dynamic and static SQL.

With dynamic SQL, the SQL statement to be executed is not known until runtime. At runtime, the SQL is passed to the DBMS in string form, run through the query optimizer and processed on the server. The same procedure is performed for every statement even if it is executed multiple times. Call Level Interfaces by nature always execute dynamic SQL.

With static SQL, the SQL statement to be executed is already known and optimized prior to its execution. At runtime, the requesting program only passes a designated handle to the DBMS to request execution of the SQL statement. Static SQL requires embedded SQL.

Due to the fact that the SQL statements must be processed at compile time static SQL can only be implemented in conjunction with embedded SQL. However, within embedded SQL both static and dynamic SQL can be performed.

The main differences between static and dynamic SQL are:

- Static SQL improves security. All statements are made known to the DBMS at (pre)compile time.
- Static SQL may improve performance. The query optimizer is only run once at bind time. This advantage may decrease as the database contents change and thus the optimization strategy becomes outdated.
- Static SQL requires embedded SQL

For more detailed information about this subject refer to VisualAge Smalltalk Server Guide V4.5 at Chapter 10. Developing database server applications

8.1.3. System Requirements

The Mynd frameworks embedded SQL support is based upon the SQL precompiler integration which IBM made available starting with version 4.0 of VisualAge/Smalltalk. This version supports DB2 version 2.1 and higher on the following platforms:

- OS2
- Windows NT
- Windows 95/98
- MVS

To use embedded SQL on any of the PC platforms, you need to have the DB2 development toolkit (or DB2 standalone) installed. To use the MVS option, you need the MVS Server Development components.

8.2. Architecture

The following components implement the embedded SQL functionality:

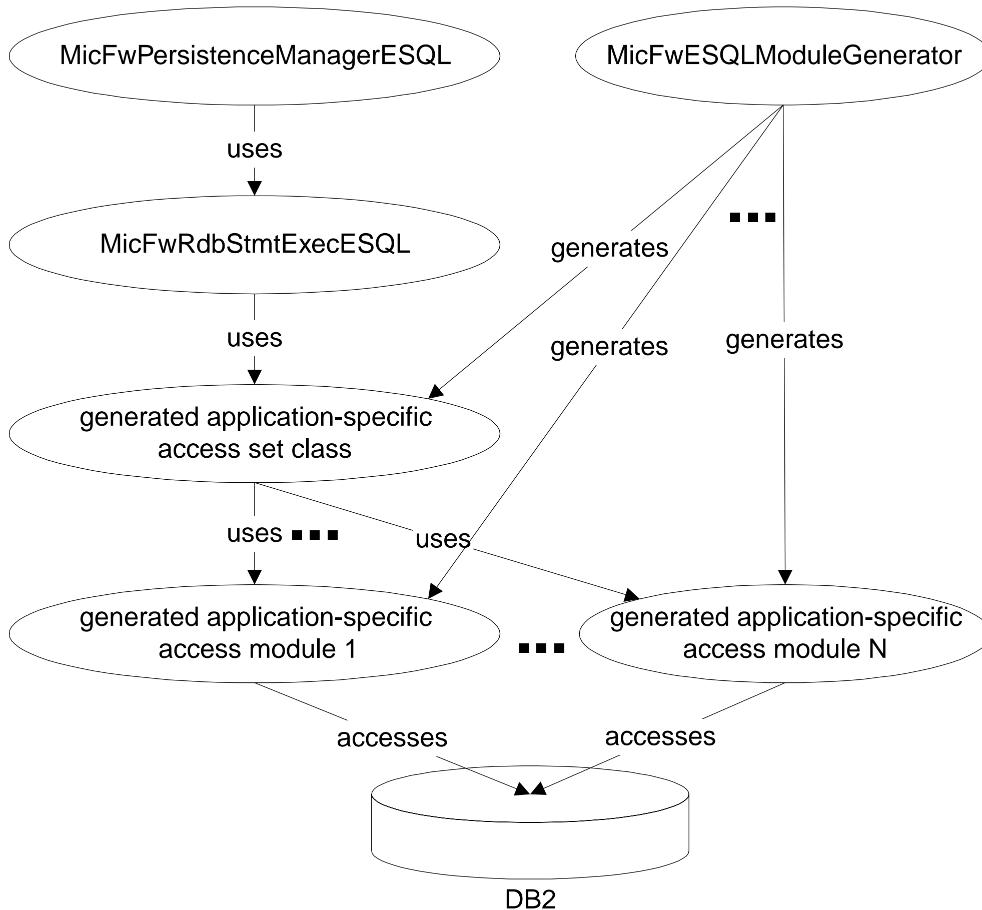


Figure 46: ESQL Persistence Components

The POM with all mappings is created as usual. Before executing in embedded SQL mode, the POM is precompiled. In this process, an access set class (subclass of *MicFwEsqlAccessSet*) and one or more access module classes (subclasses of *MicFwESQLModule*) are generated.

For all statements that have been chosen for pregeneration and assigned query IDs in the statement registration tool, instance methods containing embedded SQL are generated in one of the access module classes. The splitting in more than one module is needed for large POMs, because the DB2-precompiler sometimes has problems with more than 1000 statement in one module. The following methods are generated for the respective statements:

statement type	esql method
INSERT	#esqlInsertXX:
UPDATE	#esqlUpdXX:
DELETE	#esqlDeleteXX
SELECT	#esqlOpenXX:, #esqlFetchXX:, #esqlCloseXX

Table 15: ESQL methods generated for SQL statements.

where “XX” designates the internal query ID that is assigned to each statement. All modules from a POM are combined in the generated access set class, which also holds information about the connection and the package specification.

At runtime the persistency operations requested by the application are automatically mapped through the generated access set class to the appropriate ESQL methods by the framework. If a persistency operation does not map to a pregenerated statement with an assigned query ID, the framework attempts to execute it dynamically.

Note: The IBM embedded SQL implementation does not support dynamic SELECT statements. Therefore, in a workstation environment, dynamic SELECT statements are routed to the CLI interface. In a MVS environment, however, these will result in a runtime error since CLI is not available there

8.3. Using Embedded SQL

Development with embedded SQL normally takes place in the following steps:

- create mappings for the persistent classes in the application (if the database does not exist, also create the database)
- application development and testing phase
- conversion to embedded SQL with subsequent testing
- deployment

8.3.1. Mapping for Embedded SQL

The mapping process takes place as usual. A POM can be generated from an existing object model, classes can be generated from an existing DBMS schema or existing classes can be mapped manually to an existing schema. The POM class can either be *MicFwPersistenceManagerOdbc* or *MicFwPersistence-ManagerESQL*. An Odbc-POM must be converted into an ESQL-POM for using embedded SQL (seeCon-verting an existing POM to embedded SQL). If schema import is necessary, an ODBC data source must be available in either case.

Queries can only be executed statically if they are registered with the POM. Therefore it may be desirable at this point to register queries, too.

8.3.2. Application Development and Testing

During the initial application development and testing phase the embedded SQL aspect can be neglected almost completely. The only point that needs to be kept in mind is that object queries can only be executed statically if they are registered with the POM. Therefore it is advisable to register queries early on, using the registration tool, and avoid coding queries directly. For queries that are not registered and thus not pre-compiled, the framework will try to execute them dynamically.

Since the precompilation process can be quite lengthy with large POMs, it is advisable to defer it until the mappings for the application have proven to be stable. During this time, the POM can be run in dynamic mode via ODBC.

8.3.3. Converting an existing POM to embedded SQL

If you have used a POM of class *MicFwPersistenceManagerOdbc* (or a subclass thereof) during develop-ment you need to convert it to class *MicFwPersistenceManagerESQL* before precompilation. This can be done with the following script:

```
| newPom odbcPom |
odbcPom := MicFwPersistenceManagerOdbc named: 'MYNAME'.
newPom := odbcPom copyAs: MicFwPersistenceManagerESQL.
newPom browsePom.
```

The last line opens the STOPF tool on the new POM.

Note: The database schema of your ODBC POM must not contain columns whose data type has no coun-terpart for native DB2. Otherwise you will encounter errors when using the copyAs: method to convert your ODBC POM into an ESQL POM.

As soon as you have a *MicFwPersistenceManagerESQL* POM, you can start the precompilation from within the STOPF tool. See the description from the Tools chapter. During precompilation, a statement executor class is created.

Note: when precompiling large POMs, it may be necessary to increase the log file size for your database

8.3.4. Deployment

During precompilation, you can choose to immediately bind the package against the DBMS and to create a bind file. If you choose to create a bind file, it is stored in the directory where the image resides. This file needs to be used to bind the application if the development database differs from the runtime database.

8.3.5. Connecting the POM

There are the following connection modes for ESQL POMs:

- ESQL (static) connection. This connection is used by the embedded SQL methods at runtime.
- CLI (DB2 call level interface) dynamic connection. This connection is created automatically whenever dynamic SQL calls are made (e.g. for unregistered queries). It can only be active in workstation envi-

ronments (not under MVS).

If you establish a connection in the STOPF or interactive SQL tool, it will be created as a CLI connection. The following APIs are available:

- *#connectDataSource*. (Runtime) Re-establish the connection that was stored during the precompilation process.
- *#chooseDataSource*. (Development) Explicitly create a dynamic connection via ODBC. Used for development tools.
- *#connectDynamic*. (Runtime) Explicitly create a dynamic connection via CLI on an existing ESQL-connection
- *#connectTo: dbName user: userid password: password.* (Runtime) Connect to the DBMS with the specified attributes
- *#connectTo: dbName user: userid password: password mvsSsid: ssid mvsPlan: planName.* (Runtime, MVS) Connect using the specified attributes. This call is portable across all platforms, including MVS/CICS and MVS/native.

8.3.6. Mixing Dynamic and Static SQL

It is possible to use both dynamic and static SQL with the same POM. Static SQL will be used for those operations that were registered and assigned query Ids before precompilation. For other operations (e.g. statements executed through the *#executeSQL:* API and unregistered queries), dynamic SQL calls are made. There are 2 ways of executing dynamic SQL:

- *execute immediate* calls within embedded SQL. If the appropriate option was chosen during precompilation, an *executeImmediate:* method will have been generated in one of your access set module classes. This method is used to execute non-SELECT statements. Parameters are embedded within the SQL string, using the *asESQLString* method. *Execute immediate* is also available on MVS platforms.
- *DB2 CLI calls.* This path is chosen for SELECT statements and if *execute immediate* was not configured in your POM. It is only available in workstation environments (Windows, OS2). Parameters are converted bound through the appropriate APIs.

Note: If embedded and dynamic SQL are mixed according to (2.), it may be necessary to increase the *applheapsz* parameter for your database

8.4. Tool Support

Before an ESQL POM can be executed it must be precompiled. As described in the previous section precompilation involves generating methods for all registered statements and running them through the appropriate precompiler for the target platform.

Note: For large POMs it may be necessary to enlarge the DB2 transaction log file size to avoid “the transaction log is full” errors during precompilation. This can be done from the DB2 database director tool in the database configuration dialog. The size used for internal testing with the PFW is 350

Before beginning the precompilation process the statements that should be generated must be registered and assigned query Ids. This is done in the Statement Registration Tool which can be accessed from within the STOPF tool. Only those statements that are registered and assigned id’s are generated and precompiled. If, upon opening the precompiler dialog, registered statements without query id’s are found, a warning message is issued.

After registering statements you can enter the precompilation dialog by selecting “Generate & Precompile Embedded SQL” from the “Manager” menu in the STOPF tool.

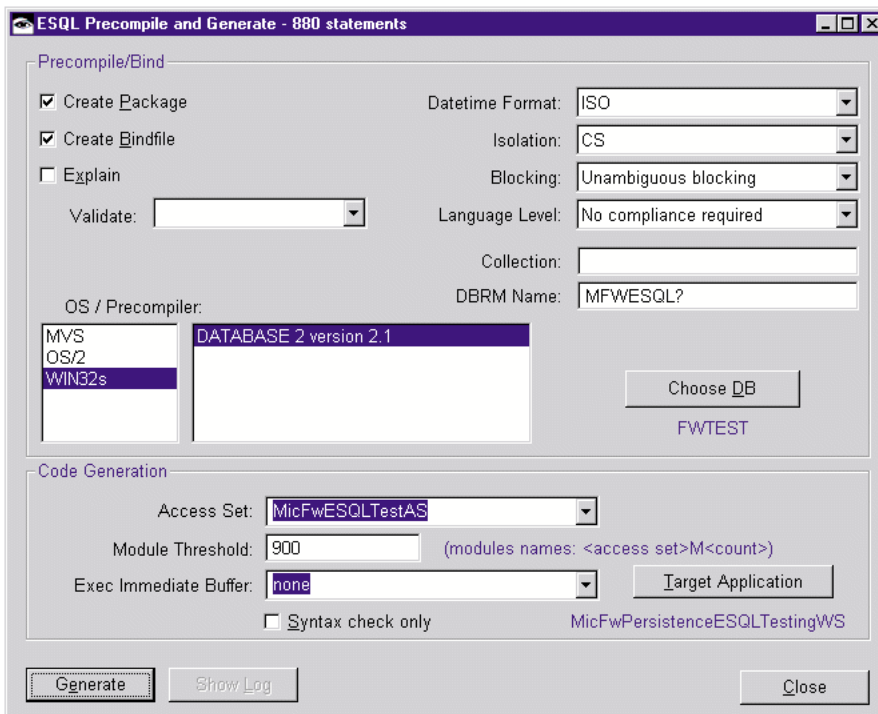


Figure 47: Precompilation Dialog

8.4.1. Precompilation and Code Generation Options

The precompilation dialog shows 2 groups of options:

- Precompilation and bind options. These options correspond to the options provided by the DB2 precompiler and bind utility.
- Code generation options. These options configure the code generation process
- Precompile and bind options that can be set:
- OS / Precompiler. The list in the middle of the dialog is used to configure the precompiler version used for specific target operating system platforms. If no choice is made in this list the default precompiler is chosen. Currently, alternative precompilers are only available under MVS. For further information on the other precompilation and bind options please refer to the DB2 documentation.
- DBRM Name. (DBRM = Database Resource Module) This is the package name of the bind file. If multiple modules have been computed (based on the threshold), the DBRM name must be expanded with a counter to generate the names. This is done automatically either by appending the number to the given name, or by replacing the first occurrence of the '?' character within the name. The name (excluding the '?') must leave enough digits to accommodate the counter (max namelength = 8).
- Choose DB. With this button the target database can be selected. The selected database is displayed below the button.

For further information on the other precompilation and bind options pleas refer to the DB2 documentation.

The following code generation options are presented:

- Access Set. This specifies a class name that will be used for the access set class. The access set class is used internally by the IBM ESQL implementation and should not be used directly or modified manually. It holds class methods which describe the connection and the package that are used by the precompiled methods. If an existing access set class is chosen from the list, it will only be overwritten if the connection or package parameters have changed. If a name for a new class is entered, it will be generated during precompilation
- Module Threshold. The generated Methods which contain the embedded SQL are generated in access module classes. This option sets the maximum of statements generated in one module. The setting of a threshold is optional, but should be between 500 and 900. The modules needed to hold all statements are generated during precompilation. The names of the modules are equal to the access set name with an added ‘M’ and the index of the module. The methods of existing module classes are overwritten and obsolete methods are deleted. Statements for one class are always kept together, therefore the module size may not exactly match the threshold.
- Execute Immediate Buffer. This option determines the size of the string buffer host variable which will be used for executing dynamic SQL statements. If an integer value is chosen from the list, a method *#executeImmediate:* will be generated in the statement executor. This method takes a string parameter whose size must not exceed the value specified in the option. If the value chosen is “none”, no *#executeImmediate:* method is generated. The statement executor will therefore attempt to execute all dynamic SQL class via DB2 CLI.
- Target Application. Use this button to choose the application into which the generated classes and methods will be stored. Only application that have the framework application *MicFwPersistenceESQL* as prerequisite can be choosen.
- Syntax check only. When the checkbox ‘Syntax check only’ is activated, the label of the ‘generate’ button is renamed to ‘check’. In this case only the SQL-statements are checked and no smalltalk code is generated.

The ‘generate’ button is deactivated until an OS, a precompiler, a database and a target application has been chosen and names for the access set and the DBRM has been set. After choosing the ‘generate’ button, the precompilation process will start. During precompilation, log information is written to the transcript.

Notes: If you are using a standalone version of DB2, the database must have been started before precompilation or syntax checking. If you choose the *explain* option, the appropriate explain tables (see DB2 documentation) must be accessible during precompilation.
To decrease the growing of the smalltalk library while precompilation, it is advisable for large POMs to unload the generated module classes before precompilation , so the framework creates new access module classes instead of changing the existing

8.4.2. Parameter Select String Editor

If a direct statement has been created for an ESQL Persistence Manager and has parameters in the where-clause, then the direct statement require a PSS (Parameter Select Statement) SQL-statement to detect the parameter types. The PSS is created automatically by the persistence framework during ESQL-precompilation time.

In some cases the created PSS will not be the desired statement. However, it is possible to edit the generated PSS file with the ESQL Parameter Select Statement Browser (PSS-browser).

To open the browser, select from the main menu 'MicFrameworks'>> 'PersistenceTools'>> 'Browse ESQL Parameter Select Statements...'.

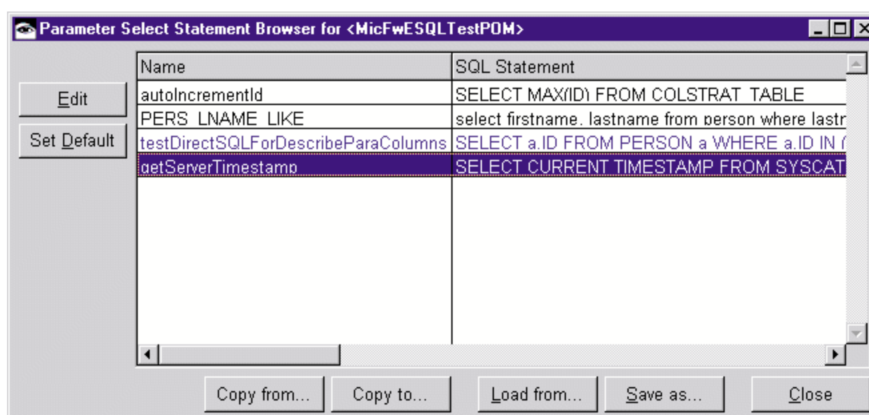


Figure 48: Parameter Select Statement Editor

The PSS-browser shows a list of all direct statements of the POM. Statements that have a PSS other than the default PSS are displayed in blue. A window for editing the PSS can be opened by double-clicking on the PSS or clicking on the 'Edit' button.Each selected statement in the list can be reassigned its default PSS by clicking on the 'Set Default' button.

The user-defined PSS's can be read from / saved to a file by clicking 'Load from...' / 'Save as...'. The user-defined PSS's can be read from / saved to a different ESQL-POM by clicking 'Copy from...' / 'Copy to...'. PPS's that have no direct statement in the POM will be ignored when loading from a file or copying from another POM.

The editor for a direct statement (opened from the 'statement and queries' browser of an ESQL-POM) shows the PSS below the direct statement. If the PSS is not user defined, the generated PSS for the current direct statement is displayed. Note: It is not possible to change the PSS in this editor. Changing or resetting the PSS to the default value can only be done using the PSS-browser.

After a POM has been deleted, the memory occupied by existing entries for user-defined PSS's should be freed. To do this, select in the menu 'micFrameworks'>>' Persistence Tools' menu item 'Clean ESQL Parameter Select Statement Storage'. This will also delete user-defined PSS's for deleted direct statements in other POM's. (Remark: When the user opens a PSS-browser, the user defined PSSs of deleted direct statements in the POM will also be deleted.)

Please note that when the POM is copied (for example, using the STOPF-tool) the user-defined PSSs are not copied. User-defined PSSs must be copied manually with the PSS-browser.

8.5. Migrating from earlier Releases

8.5.1. Lazy Initializing of Statement Descriptors

Since Release R3.3 V3.1 the framework offers much enhanced performance for embedded SQL (ESQL) through the following means:

Lazy initialization is used for the statement descriptors. This means descriptor information is created and cached when the method is accessed the first time.

The methods are accessed from a preallocated Array through indexes instead from an IdentityDictionary.

This yields the following advantages:

- for large POMs initialization time is reduced.
- since often only a small subset of the generated methods is actually used, overhead for unused methods is avoided
- access to the methods through an indexed array is much faster
- the array is preallocated with the (maximum) required size, thus avoiding expand/copy operations
- In order to achieve these benefits, the following has to be done:
- delete the method #initializeModulesFor: from any previously generated MicFwEsqlAccessSet subclasses.
- delete or unload any previously generated MicFwESQLModule subclasses
- load the new Frameworks release
- precompile all POMs as usual

The POM archival code will only need to be saved is the name of the AccessSet class was changed.

8.5.2. Multiple Module Support

Previously, the statements stored in the POM were generated into methods with embedded SQL. These methods were stored as instance methods in the associated statement executor. For this purpose, a subclass of MicFwStmtExecESQL was created automatically. The large amount of statements lead to a correspondingly large number of methods in the statement executor.

Because the DB2-precompiler has problems with large number of statements (approx. beyond 1500 statements), since Release R3.3 V3.0 the statements can be generated in more than one module (see: Precompilation and Code Generation Options Precompilation and Code Generation Options).

To migrate from a previous release, use the following steps:

- delete all your generated subclasses of MicFwStmtExecESQL.
- If you want to keep the access set classes, delete the micFwEsqlPackage class methods (optional).
- start precompilation from within the STOPF tool as usual. Specify all parameters and precompile.

9

Cross Development Support

9.1. Introduction

Together with version 4.0 of VisualAge/Smalltalk IBM introduced the possibility of creating Smalltalk applications on Host computers running the MVS operating system. Host-based Smalltalk applications can be implemented as CICS or IMS transactions or MVS native batch programs. During testing applications can be run in emulation environments (e.g. CICS/OS2). For database (DB2) access, using embedded SQL is mandatory.

Development for MVS applications is done in a cross development environment which consists of a development image and one or more passive images for each target configuration.

The Mynd Persistency Framework supports host programming by providing tools to store and precompile POMs (Persistent Object Managers) in passive images. With these tools, a previously testes POM can be transferred to a target image, deleted from a target image and precompiled within the target image.

9.2. Setting up for Cross Development

To be able to develop applications for a host environment, you need to have the XD feature loaded. To load the PFW cross development (XD) support, simply load (or reload) the following configuration maps (with required maps):

Development image	Target image
micPersistenceEmbeddedSQL Development	micPersistenceEmbeddedSQL Runtime

Table 16: Configuration maps for PFW cross development support

Note: you cannot load development support into a passive image.

After having loaded the XD support, a new submenu "*XD tools*" will appear in the "*micFrameworks*" menu in the transcript window. This submenu has the following items to choose from:

- Store POM in passive image.
- Delete POM from passive image
- Precompile POM in passive image

9.3. Using the Persistency Frameworks in an XD Enviroment

9.3.1. Application Development and Testing

Before dealing with the cross development features, your application and the persistency mappings in particular should be tested on the workstation as much as possible. For applications that use persistency framework functionality this can be easily done with any DB2 database (e.g. DB2/2, DB2/NT).

If you are using other host-specific operations (e.g. CICS, VSAM calls), your application must be tested in the appropriate emulation environment.

9.3.2. Store a POM in a target image

Before packaging and testing your application in the target environment, you need to transfer your POM to the target image. This can be done using the “*Store POM in passive image*” menu option from the “*XD Tools*” submenu. If you execute this option from within the development environment, you will be presented with a list of all currently existing native images.

By storing the POM in a target (passive) image the POM instance is added to a dictionary in the global variable *MicFwXdPomStorage* in the target image. This variable is defined in the application . When the loaded code for the application is executed, the POMs stored in *MicFwXdPomStorage* are registered to their respective classes so that they are accessble through the usual *#named:* protocol. This must be done because it is impossible to execute code in a passive image during development.

9.3.3. Precompile a POM in a target image

Once the POM has been stored in the target image, it may have to be precompiled. Precompilation must be done separately for workstation and host environments.

Note: A POM that has been precompiled under Windows NT can be used under OS/2 (and vice-versa) without need to re-precompile, provided the DBMS versions are compatible.

Precompilation is started through the “*Precompile POM in target image*” menu option. If your current image is the development image you are presented with a list of available target images. After choosing a POM from the target image, the precompilation dialog appears (see ‘Precompilation Dialog’ (page 170)). Configuration is done as usual.

Note: If you are developing for multiple environments (e.g. testing under OS/2, deployment under MVS), you may want to create separate (sub-)applications with appropriate lineups for each environment to hold the generated classes. You can then use the same names for access set and statement executor classes without overwriting the generated code for the other environment on every precompilation.

10

Tools

10.1. Introduction

This chapter describes the graphical and interactive tools which provide support for working with the PFW. The main focus is on the STOPF tool, which is used to create and edit POMs (Persistent Object Managers) and thus provides the basis for working with the Persistence Framework.

An explanation of the meaning of the acronym STOPF would appear to be appropriate at this point: It originates from the name "SmallTalk Object Persistence Framework", which initially comprised the complete Persistence Framework. In the course of time, the definition narrowed to the interactive tool whose title bar contained the acronym.

10.2. The STOPF tool

10.2.1. Overview

The use of the Persistent Object Manager with its many configuration possibilities is made much easier through the use of a graphical user interface. The STOPF tool offers this graphical user interface - all operations on POMs done using this tool can also be carried out through the programming interface of the Framework, if necessary.

Note: For most cases, it is recommended to carry out all configuration changes using the STOPF tool. The following groups of functions are supported:

- Creating and deleting a POM
- Dumping, exporting and storing a POM
- Setting the configuration of a POM
- Editing mapping and relationships
- Editing a database schema
- DDL export and import
- Pregenerate and register statements and queries

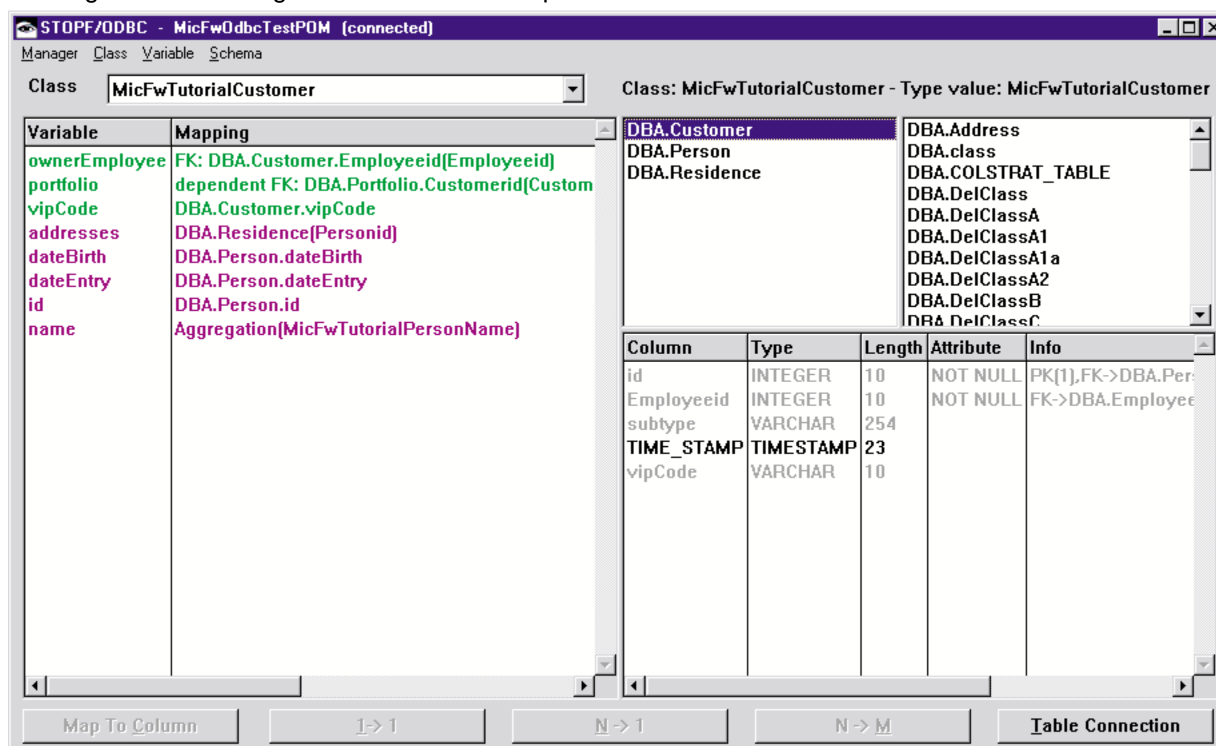


Figure 49: Main view of the STOPF tool

These functions will be explained in more detail in the following sections.

POMs can be identified by name. In programs, the ***MicFwPersistenceManager class>>named:*** class method can be used for identifying and obtaining a named POM. The POM can be stored in the class instance dictionary under the given name by using menu item *Manager→Store Globally as...*. If the POM already has a name, it can be stored by selecting menu item *Manager→Store Globally.*

10.2.2. Browsing a POM

The STOPF tool can be opened for an existing POM by selecting the Transcript menu item *micFrameworks →Browse Persistence Manager...*. You can also browse a POM within STOPF with the menu item *Manager→Choose...*.

These menu items open a POM chooser dialog in which all existing POM instances and POM initializer classes are listed. When you are within STOPF only POMs that are compatible with the already open STOPF browser are shown, e.g. when you have opened the STOPF on an ODBC-POM, you choose an ESQL-POM within STOPF.

Note: Browse Persistence Manager shows only ”tool-POM” instances, e.g. POM instances that can be browsed. RT-POMs cannot be edited or browsed with the STOPF tool, and thus do not appear in the list.

(refer to 'Run-time POM' (page 121) for details about RT-POMs.)

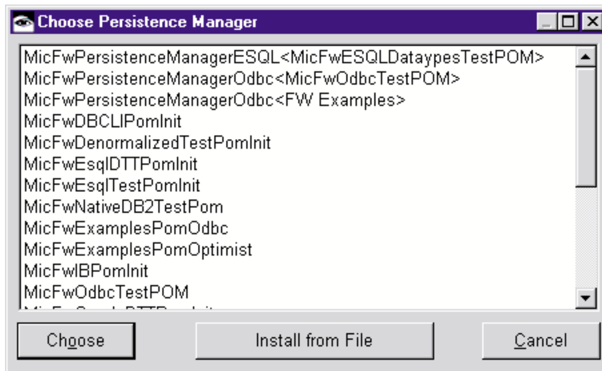


Figure 50: Browse Persistence Manager dialog

POM instances appear as upper list positions and can be distinguished from initializer classes by their printOn-format: `pomClass<instanceName>`. Only the class names of the available POM initializer classes are printed in the list.

When you want to browse a POM stored in an export file (.PST) or dump file (.POM), press the "Install from File" button and choose the file where the POM is stored in.

Note: The STOPF browser itself is an instance of ***MicFwRdbMappingTool*** or one of its subclasses, depending on the current persistence manager. The STOPF tool can also be opened on an existing POM by sending it the *browsePom* message.

10.2.3. Creating a POM

The dialog that opens when you select "New Persistence Manager" in the Transcripts micFrameworks menu is the same than for "Browse Persistence Manager" (shown in 'Browse Persistence Manager dialog' (page 185)).

The list contains all loaded subclasses of ***MicFwPersistenceManager*** that can have concrete instances. The type of class you choose creates a new POM instance and immediately opens the STOPF tool for it. The newly created instance is yet unnamed.

You can also create a new POM within STOPF with the menu item *Manager→New...* Note, that the dialog then only shows POM classes, that are compatible with the already opened STOPF browser, e.g. when you have opened the STOPF on an Oracle-POM, you cannot create a new ODBC-POM from within the browser.

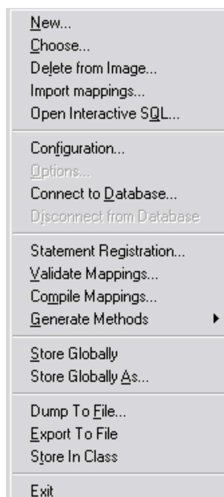


Figure 51: The Manager menu

10.2.4. Deleting a POM

It is also possible to remove a POM from the image. By clicking on menu item from within STOPF *Manager →Delete From Image...*, the user can choose which POM should be deleted from the instance dictionary. The same is possible with the Transcript menu item *micFrameworks→Persistence Tools→Delete Persistence Manager...*

10.2.5. Inspect a POM

You can open a Smalltalk-Inspector on an existing POM-instance, when you send it the message ***#inspect***. Alternatively you can open an Inspector with the Transcript menu item *micFrameworksÆPersistence ToolsÆInspect Persistence Manager...*

Note: You can inspect every POM, even those, that cannot be browsed with the STOPF Tool. RT-POMs and OLTP-Client-POMs cannot be browsed with the STOPF tool yet.

10.2.6. Storing and exporting a POM

After a persistence manager (POM) has been modified using the STOPF tool, there are different ways of storing the changes:

Firstly, the changes must be made available to the application using the POM by selecting *Store globally* in the Manager menu. This causes the changes from the work copy used in STOPF to be copied back to the POM instance that is stored globally and can be accessed using the *named:* class method. If the POM was newly created, it is thus stored for the first time. This type of storage only keeps the POM in the Smalltalk image and does not make it permanent beyond that.

The POM should also be stored in some permanent way so that it can be transferred to other images and restored after an image crash or image rebuild. There are three ways of doing this:

- Object Dump Storage (Dump to file)
- Fileout (export) Storage (Export to file)
- Class Storage (Store in class)

10.2.6.1. Object Dump Storage (Dump to file)

Using menu item *Manager→Dump to file...*, the POM is written to disk as an Object Dumper file containing the POM instance as it exists in the image.

After you have chosen a file name, you have to decide whether to dump the POM for a runtime image or for later editing with the STOPF tool. When you decide to store it for a runtime image, the POM's table schema (table descriptions and columns) will not be dumped. This makes the POM-instance smaller but you cannot open a STOPF tool for this POM or access the table descriptions of that POM in your application anymore.

The file has the standard extension **.POM* and can be loaded into the image by selecting menu item *Manager→Choose...*. as described above.

The advantages of this type of storage are:

- quick dump/install
- install can be done at runtime

Disadvantages are:

- no versioning
- stored POM may become obsolete because of version changes
- no textual representation

10.2.6.2. Fileout Storage (Export to file)

Using menu item *Manager→Export to file...* , the POM is written to disk in a text file containing executable Smalltalk statements (in "chunk format"). Opening this file, selecting all text in it and performing "file in" will recreate the POM within any target image.

Advantages:

- textual representation
- only uses public APIs, robust over version changes

Disadvantages

- no versioning
- needs development components to install

10.2.6.3. Class Storage (Store in class)

This is the preferred storage mode. This option generates an initializer class for every stored POM (subclass of ***MicFwPersistenceManagerInitializer***). Class methods are generated that recreate the mappings in the POM when executed. Methods are being generated for every table, class and relationship. A single method is generated that can be used to instantiate the POM instance from the class. POMs that are stored in this way can be browsed directly by opening the STOPF on the class ("Browse Persistence Manager", choose class from list).

You can create new initializer classes with the STOPF tool with: *Manager→Store in class*

After pressing the "New Class"-button, a prompter opens for the name of the initializer class to be created.

Afterwards the framework opens a list choice dialog to let you choose the application where the initializer class should be created in.

Remember: An application for POM initializer classes must have prerequisites to ***MicFwPersistenceRdbD*** and other framework application, depending on what kind of POM and POM-configuration you are using. When no application has those prerequisites or is ready to create a new class in it (e.g. it is not an edition or the current user does not have the neccessary priviliges), an error message appears to inform you about the problem.

Note: The POM initializer class will reference all your application classes that are mapped in the stored POM. So you should give the application with the initializer class the neccessary prerequisites, so that the mapped classes are visible from there.

Advantages:

- versioned storage
- easy to load (just like any other class/application)
- textual representation
- uses public APIs

Disadvantages:

- needs development APIs to instantiate POM

10.2.7. POM Configuration Editor

The POM Configuration Editor can be opened from STOPF using: *Manager→Configuration...* The POM Configuration Editor allows you to configure a persistence manager instance by setting options and choosing special alternatives for different persistence manager components.

A persistence manager is not a monolithic instance but consists of subcomponents for which one of several alternatives can be selected. The subcomponents that can be configured with the POM Configuration Editor are:

10.2.7.1. Database Description

The database description or *#apiDatabaseClass* of a POM is a development component that contains information on the possible data types of the underlying database and the attributes of the different types. It also contains data type to data type conversions.

Alternatives

Different database description classes which are dependent on the database actually used. You can either use a database description class based on a standard (SQL89, ODBC) or a specialized class for the type of database used for that persistence manager.

10.2.7.2. Commit Handler

The *#commitHandlerClass* of a POM is responsible for committing a persistence context to the database. The choice of a special commit handler decides which optimistic concurrency strategies are evaluated.

Alternatives

- ***MicFwRdbCommitHandler***. Commit handler for pessimistic concurrency. Does not support any optimistic strategies.
- ***MicFwRdbCmtHdlrOptimistic***. Commit handler for optimistic concurrency. During commit of a persistence context, this commit handler checks the "rows-affected" for each update statement and supports optimistic update and delete columns.

10.2.7.3. Statement Executor

The statement executor is responsible for performing any kind of database API calls (ODBC calls or others). The statement executor also handles the connection to a specific database.

Alternatives

The main statement executors available at present are ***MicFwRdbStmtExecOdbc, MicFwRdbStmtExecAbtOracle*** and ***MicFwRdbStmtExecESQL***, where a POM instance cannot use any statement executor at will, but has to use one of the compatible statement executors.

- ***MicFwRdbStmtExecOdbc***. ODBTalk: Standard ODBC 2.0 statement executor.
- ***MicFwRdbStmtExecAbtOracle.*** Statement executor for native Oracle via IBM VA Oracle feature.
- ***MicFwRdbStmtExecESQL***. Generic statement executor for embedded SQL for DB2.

The other alternatives are specialized statement executors which use different API calls to support a service. Refer to the description in the configuration editor.

10.2.7.4. Option Compare

This option decides whether objects that are implicitly or explicitly read from the database are compared with the cached representation in the image.

10.2.7.5. Generate Exists

This option decides whether the OQL translator generates EXISTS or IN clauses in SQL statements. For some databases, using in makes generated SQL statements from OQL queries execute faster, while other databases do not support IN to full extent.

10.2.7.6. Select... Configuration

Select one of the predefined POM configurations. After a POM configuration has been selected and the OK button has been pressed, the current persistence manager will be configured with the selected configuration. The predefined configurations are examples of some typical configurations for our customers' POMs.

You may add your own predefined configurations (used for your company's POM(s)) by adding a new subclass to ***MicFwPomConfiguration*** or one of its subclasses. Just re-implement the following methods in your new configuration class:

```
MyPomConfiguration>>defaultConfiguration
"Answer a dictionary with keys = componentSymbol, value = anAlternative
for the default configuration of the receiver."

"selectively overwrite default configuration of receiver's superclass now..."

^super defaultConfiguration
   at: #apiDatabaseClass put: MicFwOdbc;
   at: #statementExecutor put: MicFwRdbStmtExecOdbc;
   at: #optionCompare put:
      (MicFwPomConfigCompareOption for: true);
   at: #commitHandlerClass put: MicFwRdbCmtHdlrOptimistic;
   at: #generateExists put:
      (MicFwGenerateExistsConfig alternative: false);
yourself
```

Give your own configuration a name:

```
MyPomConfiguration>>defaultName
   "Answer the name of the default configuration"
   ^'My Standard ODBC-POM Configuration'
```

10.2.8. POM ODBC options

Menu item *Manager Options...* displays a dialog window that can be used to set various ODBC-related parameters.

The other parameters concern transaction management settings for the underlying ODBC connection. For additional details please also refer to the ODBC documentation.

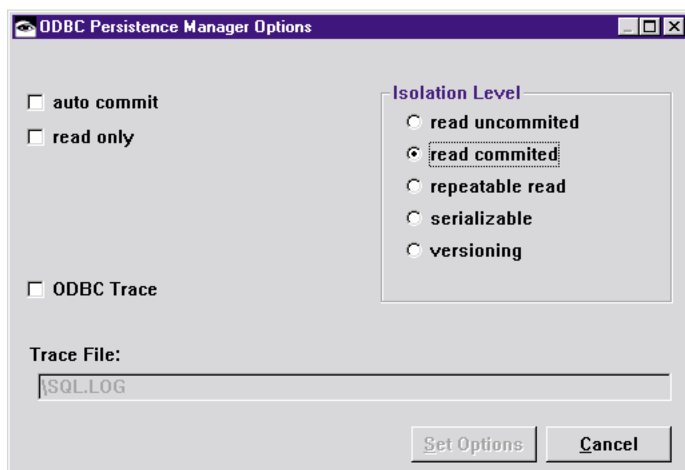


Figure 52: Persistence Manager Options

10.2.8.1. Other Settings

The remaining options on the left side of the dialog window have the following meanings:

- *ODBC trace:* Information concerning the ODBC statements currently running is output to the file listed in the text field. This function is not implemented in the current version. However, from the *micFrameworks* menu, you can activate an ODBC logger which logs the ODBC trace in a Transcript window. The Transcript window contains the standard storage facilities so that the trace information can be stored on disk.
- The *auto commit* and *read only* options are ODBC parameters that can be passed to the ODBC driver intact. You should realize that the actual implementation of these settings occurs in the driver, meaning these options may have absolutely no effect at all on the behavior of the ODBC interface, depending on the driver.

Note: The auto commit parameter should be used with caution (if at all), since it causes an implicit commit to be issued after every statement that is executed.

10.2.9. Statement Registration Tool

10.2.9.1. Tools for statement and static query registration

During runtime the persistence manager generates SQL statements for the accesses requested by the application, e.g. insertion of instances into the database, object retrieval through relationships or via object queries, if these SQL statements are not pregenerated during development time. The persistence manager caches some of these SQL statements that are generated during runtime so these cached statements remain in the POM to speed up the access the next time this statement is to be executed.

The Statement Registration tool gives the user complete control over all SQL statements that are pregenerated in a Persistence Manager. When working with conventional persistence managers (e.g. MicFwPersistenceManagerOdbc) it is not a must to care about the statement registration tool. When you decide to use static SQL POMs (MicFwPersistenceManagerESQL), all SQL statements must be registered and pregenerated in the persistence manager so that the accesses the application program requests do not cause any SQL generation during runtime.

The tool, which is called the ***MicFwStatementSettingsBrowser***, can be opened by selecting the *Manager→Statement Registration* menu in the STOPF tool.

This tool allows the user to selectively pregenerate SQL statements from base mappings, to create and modify queries (both ObjectQueries or SQL queries) and to extend relationships with order clauses. These order clauses/queries are qualified by a unique name. Registered expressions can be retrieved at runtime with one of the following API-methods:

```
registered := pom registeredExtentAt: aName.
registered := pom registeredExtentSQLAt: aName.
registered := pom registeredOrderAt: aName.
registered := pom registeredProjectionAt: aName.
registered := pom registeredStatementAt: aName.
```

This results in improved performance (no runtime SQL generation) and allows user-defined queries using static SQL.

All classes with mappings in the current persistence manager appear in the left hierarchical list box of the browser. In the lower part there are two radio button groups:

- Show types
- Show state

The **show types** radio buttons let you toggle the mode of the browser to either display SQL statements that are based on the mappings of classes and relationships (based) or to display queries, direct statements and order clauses, that have been explicitly registered and given a unique registry name (named).

With the **show state** buttons the user can toggle the display mode for based statements:

- All: Display all possible base accesses for the selected classes.
- Generated: Display base accesses for which the SQL statements are generated in this POM. These statements could have been generated and cached during runtime (in this case the Query IDs are <nil>) or could have been pregenerated using this browser.
- Pregenerated: Display base accesses with explizitly pregenerated statements for the selected classes only.

10.2.9.1.1. How to register base statements?

To register and deregister SQL statements for base accesses, select show type "based" and one or more classes in the left list. Then select the statements you want to register or deregister in the right multi-column list. Select the checkbox "***Pregenerate***" to switch the registration state.

Alternatively you can doubleclick on entries in the multi-column list to switch the registration state or you can select "*Edit*" in the "*Statements*" menu.

10.2.9.1.2. How to register named statements?

Select show type “named” and a class (for example ***MicFwTutorialEmployee***). You will see all the queries/order clauses that are already registered in this POM for the selected classes.

Select a clause/query and choose *Edit...* in the *Statements* menu in order to modify it. To create new queries, choose *New...*

Select the kind of entry you want to create (Query, SQLQuery or an order clause for one of the class' relationships that are mapped in this POM).

Note: The selection list only contains the relationships owned by/defined in the selected class - not the inherited ones! The order clause will be subsequently available for this relationship in subclasses.

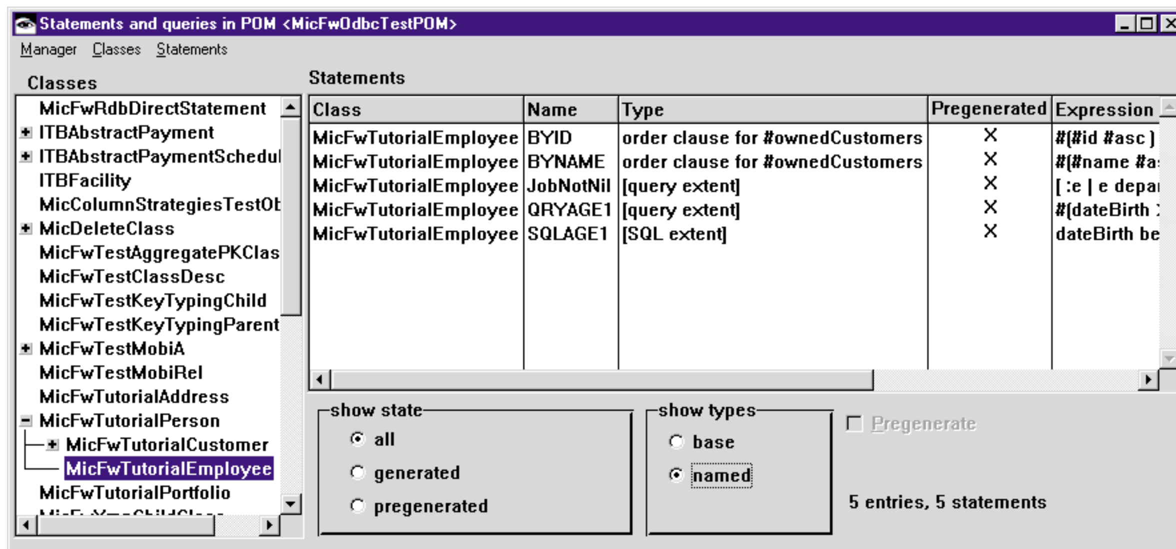


Figure 53: Statement and query registration

10.2.9.2. Direct SQL statement registration

The first item in the hierarchical class list is always ***MicFwRdbDirectStatement***. When selecting this item, the multiple column list box shows all DirectStatements that are registered in the POM’s statement registry.

API method for registering a DirectStatement to a POM:

MicFwPersistenceManagerRdb>>registerStatement: aDirectStatement as: aName.

10.2.9.2.0.1. Example

```
"** register a DirectStatement **"

| stmt pom |
pom := MicFwPersistenceManager named: 'MyPom'.
stmt := pom sqlDescriptorFor: 'DELETE FROM MyTable'.
pom registerStatement: stmt as: 'MyRegisteredStatement'.

"** usage of a registered DirectStatement **"
| stmt pom |
pom := MicFwPersistenceManager named: 'MyPom'.
stmt := pom registeredClauseAt: 'MyRegisteredStatement'.
stmt executeOnce.
```

See also API methods of ***MicFwRdbDirectStatement*** for using a DirectStatement.

10.2.9.2.1. Create/Modify a Query or SQLQuery.

Just enter a **unique** name as the registry key.

Note: The key cannot be modified after the clause has been registered. To give the clause a new key, you have to delete & create it again.

Enter an expression, either as an SQL WHERE clause such as: 'VIPCODE NOT NULL' or in object query syntax (block or array syntax). Look at the following view as an example.

An optional order clause can be entered using the array syntax. Select 'Load' to have a pregenerated 'SELECT...' statement in order to use #load, #get

and/or 'Count' to obtain a pregenerated 'SELECT COUNT(*)...' statement in order to use #answerSetSize.

Note: The 'Delete' check box is not supported at present and is always disabled.

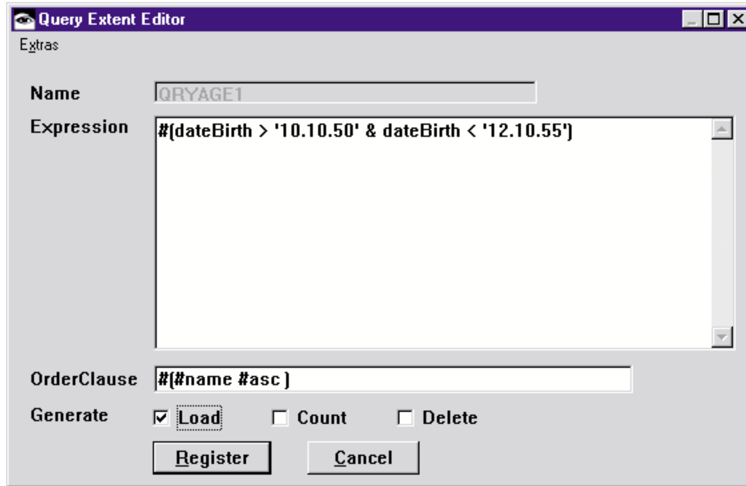


Figure 54: Query editor

10.2.9.2.2. Create/Modify registration of an order clause with a relationship

Registered order clauses can be used in all relationships that match the order clause selectors. This applies to any relationship - not just subclasses.

Because of this order clause reusability feature, the order clause to be registered with the selected relationship has to be selected from the 'Available Order Clauses' list box.

(The view shows an example for the ***MicFwTutorialEmployee>>ownedCustomers*** relationship, where target instances are ***MicFwTutorialCustomer*** objects. Available order clauses are all registered clauses including selectors that are mapped instance variables of the target class. (For example, *vipCode* and *id* are variables of ***MicFwTutorialCustomer***).

The *'OK - Order clause can be...'* message indicates that the order clause selectors are valid for this relationship.

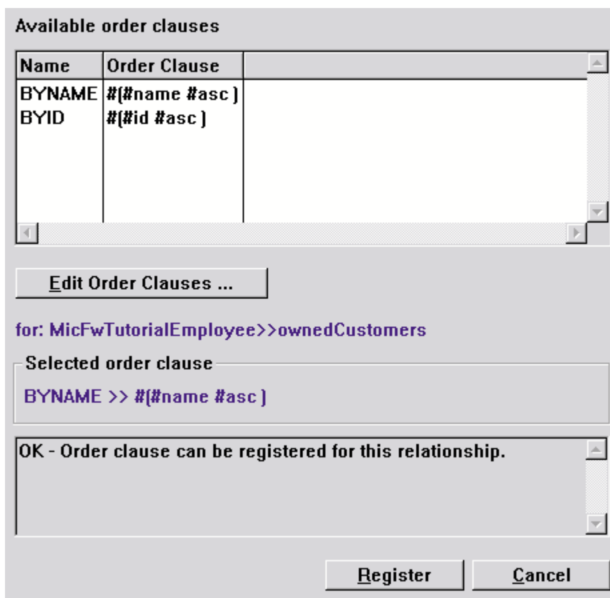


Figure 55: Order Clause Editor

The first time that you open the Relationship Order Clause Editor, the *Available Order Clauses* list box is probably empty, because no order clauses are registered in the POM. In order to register or modify order clauses (for later registration in relationship(s)), press the *Edit Order Clauses...* button, and the following

window is displayed:

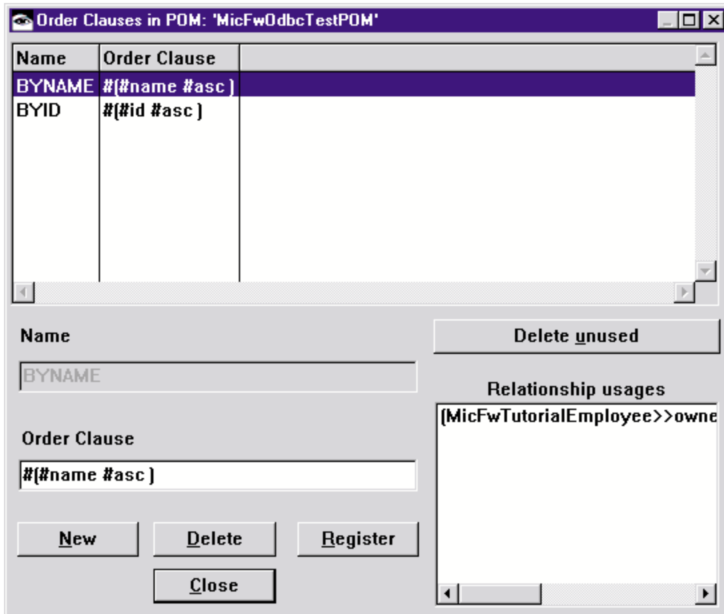


Figure 56: Order clause registration

The multiple-column list box displays **all** order clauses that are registered in the POM. You can modify them by selecting one from the list - you cannot modify the Name/PoolKey.

Every time you select an order clause, the 'Relationship usages:' list box will be updated with all relationships this order clause has already been registered with. Be careful when modifying order clauses that have usages!

Note: When you modify an order clause that is already registered with one or more relationships, make sure that the order clause still matches these relationships. Otherwise 'Please choose another order clause...' message will be displayed when the 'Relationship Order Clause Editor' is opened.

In order to clear up all the unused order clauses in the POM, press the 'Delete unused' button. Deleting a used order clause by pressing 'Delete' will also remove all registered statements for all relationships.

Other menu items

10.2.9.2.3. Manager menu

10.2.9.2.3.1. Reset base statements

Drop all generated and pregenerated base statements in the POM.

10.2.9.2.3.2. Reset all statements

Drop all generated and pregenerated base and named statements in the POM.

10.2.9.2.3.3. Compure Query IDs

Generate and assign unique query IDs for each statement generated in the POM. This is neccessary for a static SQL POM, because all SQL statements are identified by this internal query ID only during runtime.

To influence the query ID generation you can enter a *Query ID Prefix* and a *Counter Start Value.* The query IDs are symbols built from a concatenation of the prefix and the counter. The algorithm for query ID generation ensures, that all query IDs are unique, even if you entered some values there, that would cause a query ID to be used more than once.

Remember: The query IDs for statements that already have a valid query ID (e.g. a Symbol) will not change.

If you do not need unique query IDs (Symbols), you can alternatively choose *Mark as pregenerated* in the small subdialog for query registration. This would set the query ID of each generated statement, that is currently <nil> to <true> and thus mark this statement as pregenerated.

Remember: Pregenerated statement will be stored together with a POM-in-class storable or export-to-file storage. Generated statements will not be stored in such archival code. When marking all statements as pregenerated you can ensure not to loose them in the archived POM. This is important when using the Runtime POM Optimizer for a POM over and over again.

10.2.9.2.4. Classes menu

10.2.9.2.4.1. Find class

Search for a mapped class in the hierachical list using wild patterns. The class name you enter will not be

treated case-sensitive.

10.2.9.2.4.2. Select all

Select all mapped classes in the hierarchical list.

10.2.9.2.4.3. Pregenerate categories

For all statements in the multi-column list open a multiple choice dialog with the statement categories (displayed in the *Name* column of the multiple-column list) for each statement. This allows you to select some (or all) classes and generate the statement, e.g. categorized as ‚Df all load' for each.

10.2.9.2.4.4. Pregenerate with options

Opens a dialog that lets you choose **multiple** options, that your POM is used with. The framework will generate all statements neccessary to fulfill all selected options and tries to reduce the number of pregenerated statements as much as possible.

10.2.9.2.4.5. Generate common statements

Generate all (base) statements for the selected classes except those that are probably not used for abstract classes by the application.

Caution: There is no guarantee neither that these are all statements necessary to work with the POM in your application nore that these are not far to much statements that the application never requests.

10.2.9.2.4.6. Generate all statements

Generate all possible (base) statements for the selected classes. This are all statements that are shown in the multiple-column list (with show type ‚base').

10.2.9.2.5. Statements menu

10.2.9.2.5.1. Select all

Select all statements in the multiple-column list.

10.2.9.2.5.2. Browse SQL

Opens a workspace and displays all SQL statements generated from the selected statements in the multiple-column list.

10.2.9.2.5.3. Edit generated SQL...

This feature is currently not completely implemented and should not yet be used. In the next release, this will allow to modify the SQL statement string of each statement generated by the frameworks. Currently this is not possible for each kind of statement and the modifications will not be preserved when storing the POM in a class or export file.

10.2.9.2.5.4. Inspect Statements...

Opens an inspector on the currently selected statements. This is for debugging and testing purposes. When the debugger shows a collection of ***MicFwStatementCategory*** instances, you can test SQL generation and registration with the methods, common for all statement categories, e.g. ***#generate or #registeredStatement.***

10.2.9.2.5.5. Find Query IDs

Search for a Query ID in the statements currently displayed. You may enter a query ID (with no regard to case sensitivity) with wild pattern (*) in it. If the search expression you entered matches more than one query ID, a list dialog opens that allows you to choose the query ID that you are looking for.

10.2.9.2.5.6. Compure Query IDs

The same as in the "Manager"-menu, but affects the statements currently selected in the multi-column-list only.

10.2.9.2.5.7. Reset Query IDs

This <nil>s out all query IDs of the selected statements.

10.2.10. Validation, compilation and method generation

10.2.10.1. Validate Mapping

Within STOPF, you have the ability to validate the mappings of the current *Work Manager* using menu item *Manager→Validate Mappings...*. This helps you to avoid mapping or compilation problems in your POM. Sometimes this validation can provide an indication of inconsistent or inefficient mapping. It is not possible to detect all possible problems. This validation feature is extended with each version of the Persistence Frameworks as we become aware of other problems.

The Persistence Framework checks the consistency of the mappings for the current POM and writes all errors and warnings which are found into a workspace that opens following the validation process. If no errors or warnings are found, a message box is displayed containing: "Validation successful. The mappings are ready to be compiled."

Here is the list of validations currently supported:

No.	Workspace message	Description
V1	ERROR: Table <tablename> is mapped by class <classname> but has been deleted.	Check for tables that are referenced by mappings but deleted from the imported database schema.
V2	ERROR: Primary key column <columnname> in table <tablename> must be mapped for class <classname>.	Check for unmapped primary keys columns in tables for a particular class. Error V3 will also occur after this error message.
V3	ERROR: NOT NULL column <columnname> in table <tablename> must be mapped for class <classname>.	Check for unmapped columns with not null attribute in tables for a class.
V4	Suspicious: Foreign key column <columnname> in table <tablename> is directly mapped to variable <variablename> in class <classname> instead of a relationship or internal table connection.	Check if any foreign key column was directly mapped to a variable. This may be correct, but usually this is a mistake by the user.
V5	ERROR: Internal table connection foreign key name in class <classname> is not defined on primary key columns.	Check a foreign key for internal table connection is defined on primary key columns.
V6a	ERROR: FK foreign key name is used as internal table connection and as relationship mapping to classname>>variable.	Check foreign keys for table relations are used in no other relationship mapping.
V6b	ERROR: FK foreign key name is used as internal table connection and as M:N mapping to classname>>variable.	Check whether foreign keys for table relations are used in no other N:M relationship mapping.
V7	ERROR: Missing internal table connection between tables <table list> . Only one of these tables must not have an internal table connection and will be used as the master table for class <classname>.	Check for missing internal table connections for classes that are mapped to more than one table. Only one table may not have an internal table connection - this is the class master table.
V8	WARNING: Class <class name> is a subtype but has no type value defined.	This is not an error message. The intention is not to have a subtype expression if the mapped class is an abstract class.
V9	WARNING: Type value subtype value is used in classes: classes list.	A subtype is used in more than one class of the subtype hierarchy. Subtypes are normally used for unique differentiation between the classes in an inheritance hierarchy.
V10	WARNING: Mapping conflict: <variable> in class superclass name is mapped to a different table than in classname. This will cause a processing error when using a subtype mapping for polymorphic access.	Check for mapped variables that have mapping to a different table in the superclass than in the subclass. This is not allowed when polymorphic access is provided by using subtype mapping.
V11	ERROR: Missing optimistic column mapping for table <tablename> in class <classname>. Use optimistic columns either for all tables in this class or for none.	This error is detected when using optimistic (timestamp) columns for concurrency conflict detection, and if not all tables have this kind of optimistic column mapping. When using optimistic columns, each table that the class is mapped to should have optimistic column mapping. Otherwise the conflict detection cannot work for tables without an optimistic column.

No.	Workspace message	Description
V12a	No typing declared for classname>>variable.	The instance variable typing is not initialized properly or has not been saved after mapping.
V12b	No typing for column <columnname> in table <tablename>.	This indicates inconsistencies in the POMs schema.
V13	Incompatible typing between variable: classname>>variable [Smalltalk type] and column: <tablename> columnname [database schema type]	Report all non-compatible typing for all mappings. This error indicates that the class typing has been modified with a tool other than STOPF after mapping this instance variable, or the typing has not been saved using STOPF after mapping. Set a proper type for this instance variable or implement appropriate type conversion in the type converter used for this instance variable.
V14	ERROR: Table <tableName> is unknown although it is referenced by foreign key: <foreign key description>.	A foreign key in the POM's schema points to a table that is not part of the POM's schema. Remove this foreign key or import the missing table.
V15a	WARNING: Table <tableName> has no primary keys defined.	A table has no primary keys defined. This is a warning, because there are no classes mapped to that table yet. Declare primary keys before you map classes to that table.
V15b	ERROR: Table <tableName> has no primary keys defined but has classes mapped to it.	A table has no primary keys defined. There might be processing errors when working with classes mapped to that table. The Framework is unable to insert instances mapped to that table. The table needs a primary key definition so that the instances can have a persistence identifier.
V16	ERROR: Column <columnName > referenced by foreign key: <foreignKey> does not exist in table <referencedTable>.	A column referenced by in a foreign key does not exists. Remove this foreign key and declare it again. This may happen after removing a column without removing foreign keys pointing to it.
V17+	ERROR: Foreign key: <foreignKey> does not match the primary key of table <referencedTable>.	A foreign key must match the whole primary key of the referenced table (all primary key columns in the right sequence). Remove this foreign key and declare it again. This may happen after the schema has been reimported from the database and some table definitions have changed.

Table 17: List of validations currently supported

Note: Deletion of instance variables that are mapped in a POM will make this POM inconsistent. The current version of the Persistence Framework cannot detect or resolve this problem automatically. Please make sure that you have removed all mappings for variables that you want to delete.

10.2.10.2. Compile Mappings

Before a POM can be used for loading or storing persistent objects, it must be compiled. Compilation of a POM is a process that (re)builds special inner mapping structures that are required for storing or reading persistent objects. A POM has to be recompiled every time a mapping has changed.

Note: Normally a compile process runs without errors, but if the POM's mappings are inconsistent or incomplete, there may be walkbacks during the compilation process. If the validation process described above reports no errors, the compilation process should also run without problems.

Select *Compile Mappings...* from the *Manager* menu in order to compile the POM's mappings. The API method called to compile a POM's mappings is: #compileMappings.

10.2.10.3. Generate description and accessor methods

Variable typing also takes place when classes are mapped using STOPF. Whereas the mapping are stored in a different place than the classes in the POM ClassMappings, the typing information is stored in a class method of the mapped class, called the *createExtendedDescription* method (see [OBFW] for details).

STOPF remembers all classes for which typings (typed instance variables or relationship mappings) have been modified using the tool during a mapping session. Before closing STOPF, you should generate the class description and the accessor methods for these modified classes. The best method is to select *Generate Methods→ Both...* instead of consecutively generating the class description and the accessor methods. STOPF subsequently removes the classes for which both kinds of methods have been generated from the list of modified classes.

10.2.11. Editing mappings and relationships

The main task of the STOPF dialog window is to read the actual object mappings for the database(s). The class-related operations can be found in the *Class* menu, and the operations related to instance variables are in the *Variable* menu. The central element held together by the mapping information is the *ClassMap*, which can be obtained using

```
MicFwPersistenceManagerRdb>>mappingFor:
```

or

```
MicFwPersistenceManagerRdb>>mappingFor:ifNone:
```

for example:

```
classMap := (MicFwPersistenceManagerOdbc named: "MyName")
mappingFor: MyClass.
```

10.2.12. The Class menu

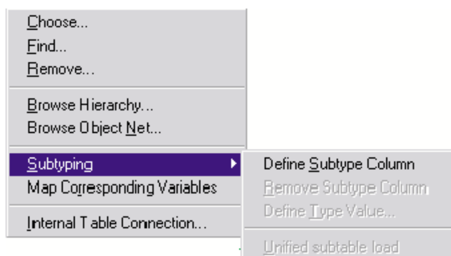


Figure 57: The Class menu

By selecting *Class→Choose Class...,* you can select from a list one of the classes that are already registered with the POM, or select any persistent class(es) from the dialog window that appears after clicking on the *Any Class...* button. The typing, relationship and transaction information can be obtained from the *extended description* of the class and can be displayed for each of the instance variables in the *instance variables list*. Previously mapped tables are carried over to the *Table List* in the STOPF dialog window. Classes can be removed from the POM using *Class Remove Class....*
The ***MicFwPersistenceManagerRdb>>removeMappingFor****:* method is called to remove the mappings of the selected classes from the POM. The menu *Class Browse...* opens a Smalltalk class browser on the selected class.

10.2.12.1. Subtyping

Class hierarchies can be represented across several tables. On the other hand, this also means that a table can store the instance variable values of objects in various classes that participate in inheritance relationships. *type discriminator columns* are used to assign the correct class to an instance during a retrieval. See chapter 'Type Discriminator' (page 77).

A specific table column can be specified for use as the subtype column for classes that contain subclasses using *Class→Subtyping→Define Subtype Column*. The table and the column to be used must be selected from the *Map Table List* or the *Column List*. Any data type can be used, but the use of a short *CHAR*, *VARCHAR* or INTEGER data type column is recommended.

After having selected a derived class, the user can enter a constant value with the corresponding data type in a dialog window which will then identify the type by clicking on *Class→Subtyping→Define Type Value...*. The value entered in the prompter will be converted to the data type of the column. This target data type is

shown in brackets in the prompter, such as '*Cust*' as a string constant or *4* as an integer constant for use in identifying the *Customer* class.

The unambiguous association of constants to classes is checked by STOPF when the POM is validated.

A subtype column can be removed using *Class→Subtyping→Remove Subtype Column*. This means that the unique column that has been used as a subtype column is released from specific mapping. STOPF removes this mapping information and all type values defined for the mapping from the POM being edited after receiving confirmation from the user.

When you check on the menu item *Class→Subtyping→Unified subtable load* the load strategy for instances of the selected class will be set accordingly. For details about the different load strategies refer to 'Unified subtable load' (page 75).

10.2.12.2. Internal Table Connection

When instances are spread across several tables, as is typical for class hierarchies, the POM used to process the variables needs information on table connection possibilities. These mappings, which are needed internally by the POM, are the *internal table connections*. They are based on the foreign key relationships between the tables in the database across which the instance variable is spread. See 'Classes Spanning Several Tables' (page 78).

The tables are always processed in the direction of the foreign keys, i.e. the class to be edited is selected first, and then the table containing the appropriate foreign key references to another table (with additional mappings of the same class) is selected. By using *Class Internal Table Connection...*, the user can finally create the connection by selecting a foreign key from the list and confirming the choice with *Connect Tables*.

It is also possible to delete an existing internal table connection using the same procedure. The user ends up in the same dialog window as above and can delete the connection by clicking the *Delete Connection* button.

Note: You can use a foreign key either with an internal table connection or with relationship mapping.

10.2.13. The Variable *menu*

Figure 58: The Variable menu

This submenu contains all items pertaining to the mapping of instance variables, i.e. direct mappings to columns or those used for managing relationships.

10.2.13.1. Instance Variable Mapping

Instance variables are mapped directly to table columns using *Variable→Map To Column*. You should check to make sure the data types are compatible (see [OBFW]).

Relational databases and the SQL standard only support a small number of scalar data types for table columns - see, for example, [ODBC2.0], Appendix D, Page 617.

Any direct instance variable mapping can be removed from the table column using *Variable Remove Mapping*. The data type information remains untouched.

Menu items *Variable→Set Type...* and *Variable→Remove Declaration* correspond to the typing of instance variables according to ***MicFwObjectDescription / extendedDescription*** (see [OBFW], Chapter 3.2 Net Browser) and realizes the setting up and deleting of types. However, deleting a type leads to any mapping information also being lost if such information existed (but only after the user has confirmed the deletion).

10.2.13.2. Foreign-key-based relationships

With the exception of an N:M relationship, relationships are created starting with the side of the relationship where the foreign key is, i.e. for a 1:N relationship the mapping starts with the class on the N side. The desired relationship can be constructed using *Variable Persistent 1→1 Relationship...* or *Variable Persistent N→1 Relationship...*. The subsequent dialog windows that appear allow various parameters to be set

for the current relationship. When the relationship is created, it should be ensured that the target instance variables of the class on the opposite side of the relationship - if there are such variables - are mapped accordingly.

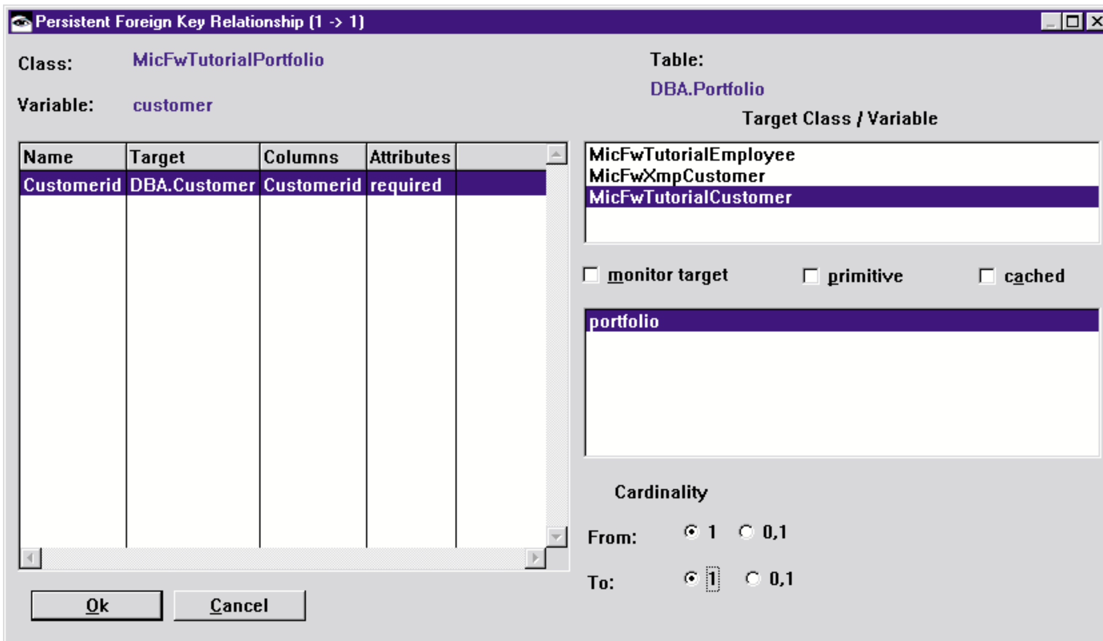


Figure 59: The 1:1 Relationship dialog window

All unmapped foreign keys in the selected table appear in the list of *foreign keys* in the 1:1 relationship dialog window. After selecting a foreign key, all classes that are mapped to the referenced table appear in the *Target Class* list. After selecting a class, the names of instance variables belonging to the class that have not yet been mapped appear in the *Target Variable* list. The selected variable represents the other direction of the relationship.

The *primitive relationship* check box is not marked yet, i.e. it is assumed that this is a standard two-sided relationship. If this box is checked, a one-sided relationship will be created; the *Target Variable* list is empty, since no instance variables are required for the reverse direction of the relationship.

When a relationship tries to get its target object, this normally results in a database call (SELECT statement). The user may mark a relationship in such a way that resolving the relationship does not produce an SQL statement, but takes place within the Smalltalk image under the control of the application. This is possible by clicking the *cached* check box. The *cached relationship* feature only works with both *primitive* and *static relationships* at present.

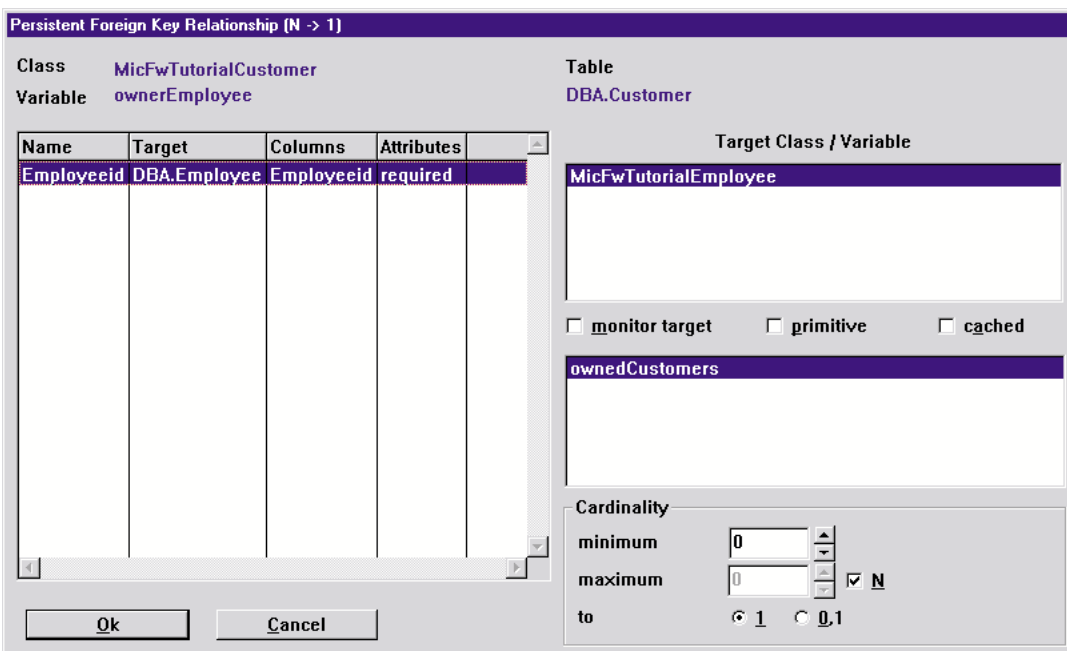


Figure 60: The N:1 Relationship dialog window

The *Cardinality* box at the bottom right of the window allows the user to specify whether an instance should exist for one side of the relationship (*From:* field = source, *To:* field = target). A NULL value is not allowed in the table column for identity or existence relationships. See 'Identifying relationship' (page 43) and 'Existence dependency' (page 43); accordingly, the dialog window does not allow the user to activate the *0,1* button in this case.

The dialog window for N:1 relationships only differs from that for 1:1 relationships in that the *Cardinality* box is displayed at the bottom right. The cardinality for the :N side of the relationship can be specified here. The cardinality can either be given exactly (enter the values in the *minimum* and *maximum* fields, do not activate the *N* check box) or it can be specified as having an unlimited maximum value (enter the value for the *minimum,* activate the *N* check box). The lowest minimum value that can be entered is 0 for all cases, because the foreign key relationship and the resulting existence dependency work in the opposite direction.

Note: For example, in one possible model, a customer is assigned to exactly one employee. The customer is therefore dependent on the existence of an employee, i.e. if an employee is deleted from the company upon quitting, then his or her customers must be assigned to another employee (or the customer must be deleted, which is not exactly the most reasonable thing to do).

On the other hand, an employee can attend to any number of customers; a special case of this being when the employee is not assigned any customers at all.

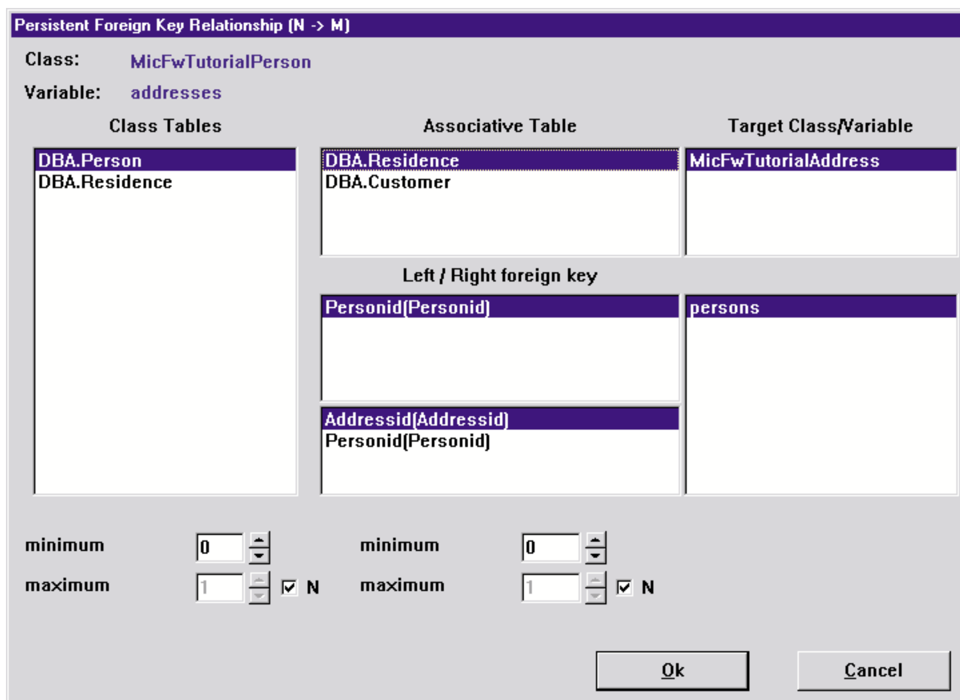


Figure 61: The N:M Relationship dialog window

Creating an N:M relationship in a relational model requires the use of an associative table in which the table's foreign keys refer to the tables to be connected. This information can be found in the middle of the N:M relationship dialog window (*Associative Tables*, *Left / Right Foreign Key*). The dialog window appears after an unmapped instance variable of the class being edited has been selected, and menu item *Variable→Persistent N:M Relationship...* has been activated.

In an almost symmetrical manner, the mapped tables of the class being edited appear on the left side of the dialog window, and all possible target classes and instance variables appear on the right side. Connections are established by selecting the appropriate elements from left to right and from top to bottom.

There are two control groups (*minimum*, *maximum*, *N*) below the lists in the *Cardinality (N:M)* box. The cardinality of the corresponding side of the relationship can be set here in the same manner as described for N:1 relationships.

Warning! An important factor for coordination between the POM and the database is synchronous (as synchronous as possible) mapping to the database, especially when mapping relationships. If the RI rules in the POM and the database do not match, then inconsistent error messages or exceptions may be displayed by the Framework or the database management system at run time

10.2.13.3. Aggregate relationships

For objects with complex (non-scalar) instance variables, you have the opportunity of storing the instance variables in a table. This is usually desired when the identity of the instance variables - when viewed as

objects - only appears in connection with the embedded (*aggregated*) object in the Smalltalk sense, otherwise it is irrelevant.

Example: The name of a customer or another person are combined to create a complex object comprising a last name and a first name. This name-object doesn't have its own identity, but only appears embedded within a persistent person object.

For combinations such as this, an *Aggregation Editor* is opened using menu item *Variable Aggregated 1->1 Relationship...*. If the enclosing class is not already mapped the STOPF tool opens a dialog box to ask if the enclosing class should be mapped now. Answer true to continue with the aggregation mapping.

In this version, a 1:1 relationship editor opens to set the relationship underlying the aggregate mapping. The editor is similar to the relationship editor provided by the Object Behavior Framework. Select a target class for the relationship and a variable for the other relationship side or press the "Primitive" button. Press "Continue.." to close the relationship editor and to open the aggregation mapper dialog window shown in Figure: 33 Aggregation editor.

This editor allows the user to assign complex instance variables to the same table as the scalar instance variables for a class. If you want to aggregate columns from a table that has no mappings to the enclosing class, you should select the table in the STOPF dialog before you press the Aggregated 1->1 Relationship... button. All available columns from the selected table will also appear in the right aggregation editor multi column listbox.

In object modeling (e.g. OMT according to Rumbaugh), we speak of *aggregations*. These correspond to a special kind of relationship (a*ssociation*) between classes that are a part of a whole. However, at object net level these are actually relationships. Note that these relationships are not database relationships based on foreign keys.

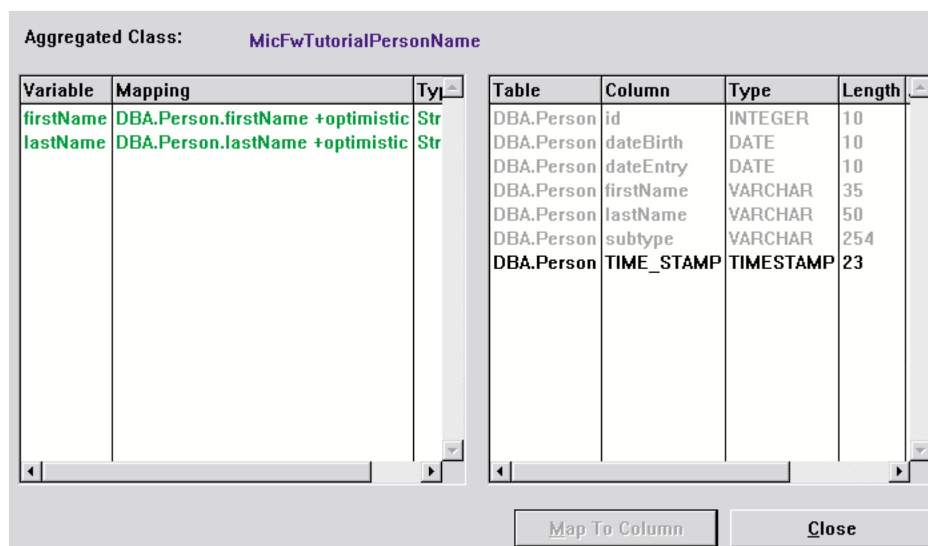


Figure 62: Aggregation editor

Instance variables of the aggregate class are mapped to the columns of a table in the *Aggregation Editor*. On the other hand, the *Aggregation Editor* can also be used to delete the mappings of an existing aggregation (by choosing the popup menu of the left listbox *Remove Mapping*).

10.2.14. User-defined relationship

The STOPF-mapping tool was extended to support the mapping of user-defined relationships (= non-standard relationships).

When a user wants to map a user-defined relationship, He has to choose a class.

Click on the variable in this class that should contain the user-defined relationship. The variable must not contain any mapping in the current persistence manager and should not already be typed.

Select menu item *User-defined relationship* in either the popup menu or the *Variable* menu. The following

screen appears:

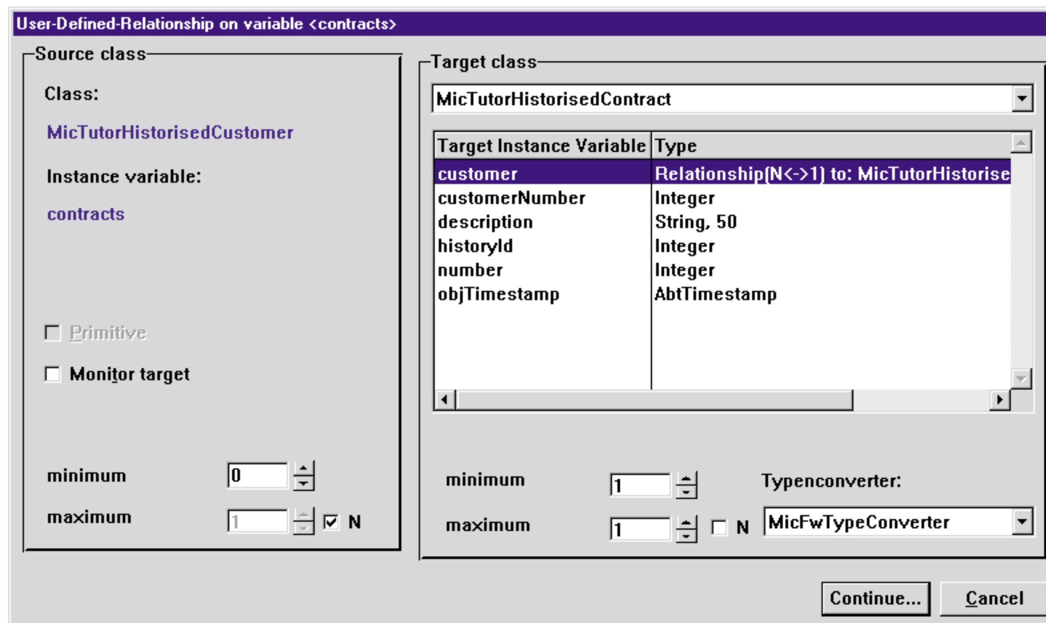


Figure 63: Relationship editor for user-defined relationships

This editor is similar to the now familiar relationship editor that is part of the "Object Net Browser".

Choose the target class and the target variable, and edit the cardinalities.

The relationship (for the object behavior framework) will be declared when the user presses the "Continue..." button.

The relationship has now been declared. All the following dialog windows enable the user to provide the mapping information associated with this relationship.

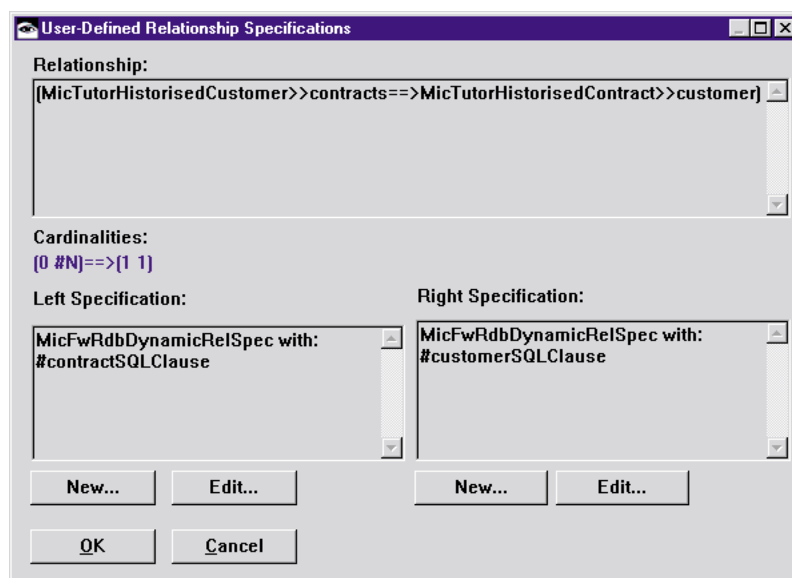


Figure 64: Editor for user-defined-relationship mappings

The above figure shows all the information that is available so far:

- The relationship itself, in the "Relationship:" text field
- The cardinalities
- The current left-side and right-side specification (or the word *none* if none has yet been specified).

Press the "OK" button to declare the mapping in the persistence manager.

To create a new specification, press the "New..." button under the corresponding specification.

Choose the type of specification to be created:

- **Dynamic.** A method selector is specified at development time. At runtime, the specified method is sent to the relationship's source object. It should return a ***MicFwRdbSQLClause*** that describes the complete WHERE clause.
- **Static SQL Clause.** An SQL clause (***MicFwRdbSQLClause***) describing the complete WHERE clause

is specified at development time.

- Static Object Query (not yet implemented) OR press the "Edit..." button to modify an existing specification.

10.2.14.1. Dynamic specification editor

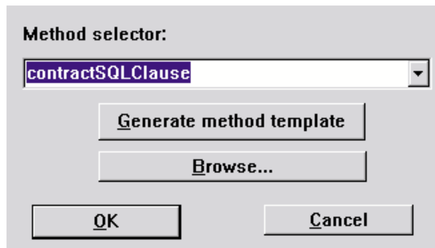


Figure 65: Dynamic specification editor

This dialog window is used to specify a method selector. This method is called each time the Persistence Framework tries to resolve the relationship. The mapped instance has to implement this method in such a way that an instance of ***MicFwRdbSQLClause*** is returned.

The *combo box* shows all the method selectors that already exist in the mapped class (these are instance methods with none or one parameter). The user may choose one of these methods or enter a new name in the combo box's entry field. For details about the dynamic callback method refer to ‘Dynamically defined user-defined relationships’ (page 82).

The *Generate method template* user button creates a method template for the dynamic method selector if no such method exists. The created method already includes a comment and shows the kind of result that has to be returned.

10.2.14.2. Static SQL clause specification editor

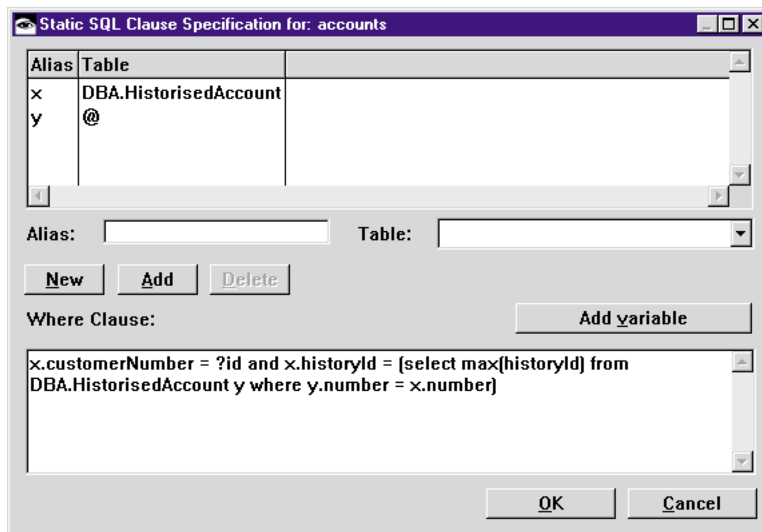


Figure 66: Static SQL clause specification editor

The Static SQL clause specification editor allows you to specify the ***MicFwRdbSQLClause*** for resolving the user-defined relationship in a convenient way.

(See ‘Class MicFwRdbSQLClause’ (page 84) for a description of the usage and further features of the ***MicFwRdbSQLClause*** class.)

The upper multi-column list box in this editor shows all the tables and their aliases that are used in the <tablelist> of the user-defined relationship SELECT statement: (e.g. SELECT [something] FROM <tablelist> WHERE <whereclause>). The user may add new tables to the statement by pressing the "New" button, choosing one of the tables in the "Table" combo box containing all the tables that are

- not already used in the SQL clause
- known in the POM’s database schema

and pressing the "Add" button.

- The alias for a table entered in the "Alias" entry field must be unique in this SQL clause.
- tableName @ is used to reserve an alias for use in a sub-select expression of the WHERE clause, as

shown in the example.

- The WHERE clause may contain any WHERE expression that will be added to the statement. The WHERE clause may have parameter markers that will be filled with current instance variable values of the specific class.
- In order to add a valid parameter marker to the WHERE clause, press the "Add variable" button and select one of the persistent mapped variables of the class. A ? will be added to the WHERE clause, followed by the selected variable name.
- The SQL clause will be generated after pressing the OK button. The editor checks the where clause in order to make sure that no invalid variable names are used.

Note: The editor does not check the correctness of the WHERE clause in a syntactical SQL sense!

10.2.15. Setting column mapping attributes

When mapping an instance variable to a single column you may specify some kinds of attributes for that mapping to deal with special database characteristics associated with that column that the Framework should pay attention to. You can open the "Attributes Editor" for instance variable mapping using the "Variable->Attributes..." menu or from the aggregation mapper (see chapter Aggregate relationships Aggregate relationships) with the "Attributes..." popup menu. The figure shows the "Column Mapping Attributes Editor" dialog.

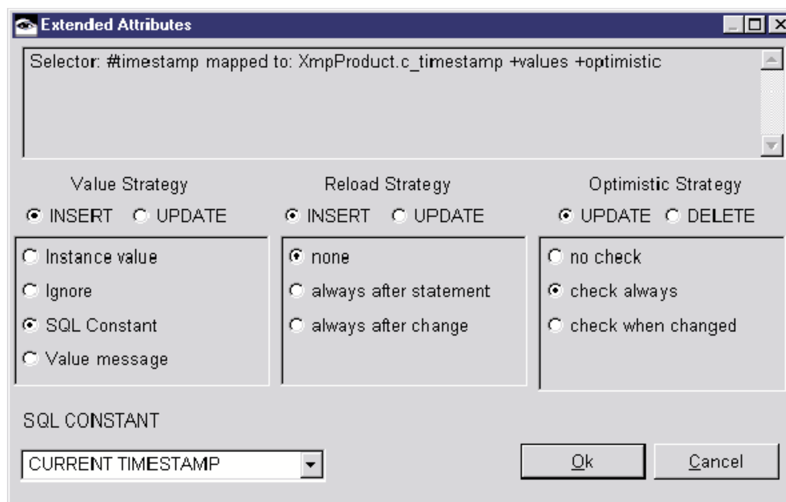


Figure 67: Column mapping attributes editor

Note: No Reload or Optimistic Strategy is available for primary key columns. Radio button will be disabled in this case.

The selection shown for the Value Strategy and the Reload Strategy depends on whether the INSERT or UPDATE radio button is selected. You can choose a Value Strategy and a Reload Strategy separately for INSERT and UPDATE operations. See 'Column Mapping Attributes' (page 88) for a detailed description of the different strategies.

10.2.15.1. Value Strategy

See 'Value Strategy' (page 88).

Value Strategies "Instance value" and "Ignore" do not have a parameter.

Value Strategy "SQL Constant" needs the SQL Constant expression specified that will be generated into the result SQL statement for the class. The user may enter any valid SQL expression in the "SQL Constant" combo box in the lower part of the dialog or choose one of the constants in the combo box's drop down list.

Value Strategy "Value message" needs the selector of the value message specified. The value message has no parameter and will be sent to the POM's current commit handler before a write operation to provide the value to be written into the database. The user may enter any valid method selector. Ensure that the commit handler you have configured with your POM understands this message. Otherwise choose a proper commit handler or implement the message in the commit handler class you are using.

10.2.15.2. Reload Strategy

See 'Reload Strategy' (page 88).

10.2.15.3. Optimistic Strategy

See 'Optimistic Strategy' (page 89) and 'Optimistic Concurrency' (page 151).

10.2.16. Editing a database schema

The database schema is an important fundamental element of class mapping. Firstly, the tables to be used must be registered by the POM being edited Secondly, the existing EI and RI rules (*entity integrity and referential integrity*; primary and foreign key definitions, existence dependencies) of the database schema should be recognized and utilized - see [CANN], chapter 22. It should also be possible to do just the opposite, i.e. to model EI and RI rules that are missing from the database in a POM "after the fact", because one may not assume that the database or RDBMS that is used supports entity integrity or referential integrity.

You may also want to modify the database schema within STOPF (add new columns or modify foreign keys) and then export the schema in order to modify your database, too.

Figure 68: The Schema menu

All the functions required for editing a database schema are contained in the *Schema* menu.

10.2.16.1. Table-oriented Functions

The entire table schema or the schema information concerning individual tables can be read into the POM being edited (*Work Manager)* using menu item *Schema→Import/Update...*. After selecting the tables from the table list, you may set some import options:

10.2.16.1.1. Import tables into POM.

If this option is set, the selected tables will be imported into the STOPF *Work Manager* and appear in the STOPF tool's *Table List*. They may then be mapped.

10.2.16.1.2. Remove conflicting mappings.

After importing the schema, all imported database columns will be checked for modifications. Existing mappings in the *Work Manager* will be removed for:

- deleted columns (columns that do not exist in the database schema),
- foreign key mappings (relationship mappings) if the primary key of the target table has changed,
- foreign key mappings (relationship mappings) if the underlying foreign key does not exist in the database schema.

10.2.16.1.3. Schema report.

If you select this option you will receive a workspace containing a report showing all the modifications, problems and removed mappings which have been found. Otherwise a message box appears when the schema import has finished.

This option can be used to re-import the database schema, e.g. if new columns have been added to the database etc.

10.2.16.1.4. Changing Tables

If a table that has been registered with the POM being edited is no longer needed, it can be removed using *Schema→Remove Table..,.* and the schema information can then also be removed from the POM.

With the help of menu items *New Table..., Edit Table...* and *Edit Column...* it is possible to enter all of the schema information manually, without using the schema import function. Remember that the column definition editor (*Schema→Edit Column...*) uses the database description of the POM to validate the possible column types, precision and scale.

Finally, the table identifier (of type ***MicFwRdbTableIdentifier***) can be changed using *Schema Edit Qualifier/Owner...*. The identifier consists of a table name, a key for the name of the owner of the table and a qualifier. The user may need to edit the data after changing the database schema. The editor that opens up allows the user to change the table qualifier and the owner of the selected table. Editing of table identifiers is not usually necessary. If it does become necessary, the *[CANN],* for example, contains an explanation of the naming concept in chapter 3.4 Names. Further information on this subject can be found in the

documentation for the RDBMS used.

10.2.16.2. Entity Integrity

Primary keys are mainly used to identify rows of data in the database. Foreign keys are the basis for creating relationships. The lack of primary and foreign key definitions in the database schema can be corrected by using one of two possible functions that are available in the STOPF tool. These functions insert the appropriate schema information in the POM being edited.

A combination of columns - in the simplest case a single column - can be declared as the primary key using *Schema→Declare Primary Key...*. Knowledge of the primary keys is important for a POM when creating the ordering between rows of data and objects. During the retrieval of a row of data, for example, the primary key is the criteria used to decide whether or not the object corresponding to the row of data has already been loaded into the image (object caching in the transaction interface). The primary key is also used by the POM to identify the row of data for the update, in order to ensure that the object created from the row of data is stored correctly in the database.

After activating this function, a dialog window appears which allows the user to specify the primary key columns. Each of the table columns can be brought into the list of primary key columns using *Select,* and removed using *Deselect*. The order of the columns in the primary key can also be changed if the key has more than one column (*Up*, *Down*).

10.2.16.3. Referential Integrity

After activating *Schema→Declare Foreign Key...,* the *Foreign Key Browser* shown below opens. From here you can create, modify and delete foreign keys of a table.

The user can delete a foreign key definition pressing the *Delete* button after selecting a foreign key. This will cause the foreign key definition to be deleted from the POM being edited.

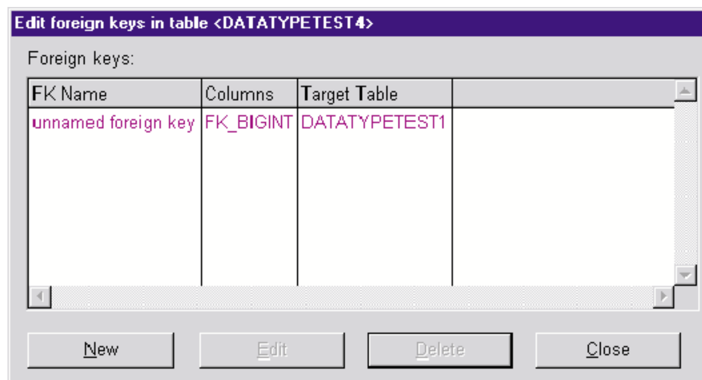


Figure 69: Foreign Key Browser

To open the *Foreign Key Editor* shown below, press the *New* button or select a foreign key in the multi-column list and press the *Edit* button. The *Foreign Key Editor* editor allows the user to edit exactly one foreign key; however, in principle any table column can be defined as a foreign key column by clicking on *Select*;

the column then appears in the *Foreign Key List* at the bottom left.

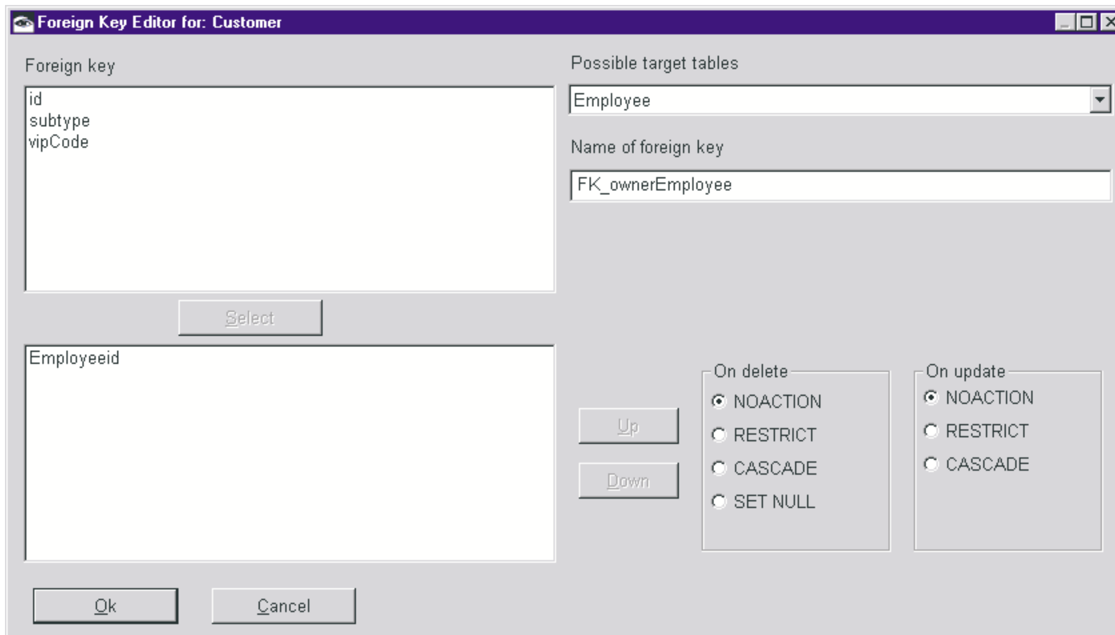


Figure 70: Foreign key editor

Foreign key columns can be removed by clicking on *Deselect*. After selecting the foreign key column, the user can obtain a selection of tables that may be used for the foreign key reference by clicking on the button in the *Possible Target Tables* drop-down list at the top right of the window. These tables are tables that have a primary key consisting of the same number of columns with exactly the same column names and data types in the same order.

The foreign key is set by selecting one of the tables and then confirming the selection in the dialog window. When specifying the foreign key, the user should keep in mind the set of possible integrity rules, which determine how dependently rows of data will react when the row of data referenced by the foreign key is changed or deleted . Some RDBMS support these *RI Actions* (*Triggered Actions*) as per the SQL standard. See, for example, [CANN] *22.8.2 Triggered Actions.*

NOACTION and *RESTRICT* normally lead to a IntegrityError, when the primary key referenced by the foreign key becomes deleted or changes. This is the default action and is supported by the framework.

CASCADE: When DBMS deletes the row containing the primary key referenced by the foreign key, the row containing the foreign key is also deleted. The framework supports this option, but you have to pay attention to the restrictions for #delete operations mentioned in 'Transaction handling' (page 98).

SETNULL: The foreign key is set to NULL. This requires there to be no NOT NULL constraints on the database column(s) in the foreign key (this rules out existence dependencies, for example). You should not use this action, because the framework currently has only rudimantary support for it yet.

In the Foreign Key Editor, you will find the options *On Delete* and *On Update* at the bottom right of the dialog window. The *SETNULL* option is not supported for *On Update* (this would not be sensible in most cases).

10.2.17. Exporting a database schema

After creating or editing the database schema in STOPF, it is possible to export the schema in order to create a new database, or update an existing one.

The schema export can be started using menu item *Schema→Export* in STOPF. First select the tables you want to export. Then the target database system, i.e. the DBMS on which the new database is to be created or an existing one modified, can be selected. After that, a new workspace window containing the generated DDL file opens.

The created DDL export file contains all the SQL statements that are needed to create the tables and foreign key relationships contained within the POM being edited.

The exported database schema can be imported by using the interactive SQL tool, for example, see chapter Interactive SQL: Interactive SQL or a tool from the DBMS which is being used.

Note: The schema export only generates CREATE TABLE statements. This means that no ALTER TABLE statements are generated after schema changes have been made. If tables with the same name already exist in the database, the appropriate table must first be deleted before the CREATE TABLE statement in

the DDL file can create the table.

10.2.18. Mapped class generation

The STOPF tool supports an editor that allows you to generate a Smalltalk class and mappings from a database table. This editor is called the "Mapped class generator". In order to open it, click on a database table in the STOPF table list and select menu item *Schema→Generate mapped class...*. (The class that is responsible for the generation is ***MicFwRdbMappedClassGenerator***).

This is the way in which both approaches are supported by the Mynd Persistence Framework:

- Generation of a POM and a database schema (DDL) from an object net (POM Generator: , Schema Export)
- Creation of a Smalltalk class with mappings from a database table (Generate mapped class)
- Manually mapping the classes to database tables using STOPF

The "Mapped class generator" adds the generated mappings to the current *Work Manager*. A complete **object net** with mappings (column-based) and **relationship** mappings (foreign key based) can be created in the *Work Manager* by generating classes for other database tables one by one. Each class gets default typings for the instance variables (according to the database column type). The necessary accessors and extendedDescription methods can also be generated using STOPF (*Manager→Generate Methods→Both).*

After the "Mapped class generator" has been opened (see 'Mapped Class Generator' (page 208)), a default class with default mapping appears in the editor window. At this time, no real class or mapping have been generated. You are advised to examine the default mappings and adapt them to requirements using this editor before pressing the "Generate" button.

10.2.18.1. Default mapping / default class

- Default class is a subclass of ***MicFwPersistentObject***.
- An instance variable with a relationship declaration for each foreign key (FK). Only primitive to-1 relationships will be generated.
- No "Internal Table Connections" are identified yet, because the default class is a subclass of ***MicFwPersistentObject***.
- An instance variable has been suggested and mapped for each non-FK column in the table.
- Default class name is the table name.
- Default variable name is either the name of the underlying foreign key or the column name.

10.2.18.2. Controls of the editor

10.2.18.2.1. Multi-column list (Variable | Mapping)

The variables to be generated and the suggested mapping to the foreign key or column.

Red lines indicate a conflict that should be resolved before pressing the "Generate" button, otherwise the red instance variables are created but not mapped.

10.2.18.2.2. Superclass

When changing the superclass, the editor checks that the variable names are unique if "Overwrite existing class" is off. If a variable name is not unique, the variable gets a unique name (by adding a number to the old variable name).

10.2.18.2.3. Class name

Name of the class to be generated. You may choose an existing class (Choose button).

10.2.18.2.4. Delete button

Deletes the selected variable; this means that the variable will not be generated or mapped.

10.2.18.2.5. Unmap button

The selected variable will be created in the generated class, but the variable will not be mapped or typed. You may choose this option if you want to map the variable later manually using STOPF.

10.2.18.2.6. Add button

After clicking this button, a prompter for entering a name for the new instance variable opens. This instance variable will be added to the generated class, but it will not be mapped.

10.2.18.2.7. Check box - Overwrite existing class

If "Overwrite existing class" is enabled, the selected class will be overwritten, which means that all instance variables other than those in the multi-column list will be deleted. If this option is not selected, the generated variables will be added to the generated target class, i.e. pool dictionaries and class variables stay the

same. With the help of this feature it is possible to map more than one table into the same generated class using the "mapped class generator".

10.2.18.2.8. Generate button

The generator creates the target class and all the mappings.

10.2.18.2.9. EntryField Variable

This field contains the name of the variable selected from the multi-column list box. This variable name can be changed. The editor ensures that the new variable name is unique for that class!

10.2.18.2.10. GroupBox/Listbox: Primitive N→1 Relationship

If you click on a relationship variable (foreign key based), this list box displays all the possible target classes for the relationship mapping.

Possible target classes are all classes that still have mapping to the table that is the target of the underlying foreign key in the current *Work Manager*.

The default selection is the first class in this list. You can select any other class in this list as the relationship target.

Remember that the "mapped class generator" only creates primitive relationships. This means that the target class does not need an instance variable for the other side of the relationship. Using STOPF, you can modify the relationships to be non-primitive or to have special cardinalities etc. after generation.

10.2.18.2.11. Listbox - Internal table connection

When choosing a superclass for the generated class that already has a mapping in the current POM, this list displays all the foreign keys that point to a table that the superclass is mapped to. One of these foreign keys should be used as the internal table connection between the tables.

e.g. : Generation of a class for EMPLOYEE table.

TutorialPerson class is already mapped.

The user selects the TutorialPerson class as the superclass → an internal table connection is now possible and, indeed, necessary.

The foreign keys in the EMPLOYEE table that point to the PERSON table are displayed in the list, and one of them can be mapped as an internal table connection.

A foreign key can either be used with an internal table connection or with relationship mapping. An entry in the multi-column list that is based on the same foreign key as the internal table connection therefore turns red in order to indicate a conflict. The user must either select another foreign key or "Unmap" or "Delete" the conflicting relationship instance variable (if the entry stays red, the internal table connection has priority and the relationship will not be mapped.)

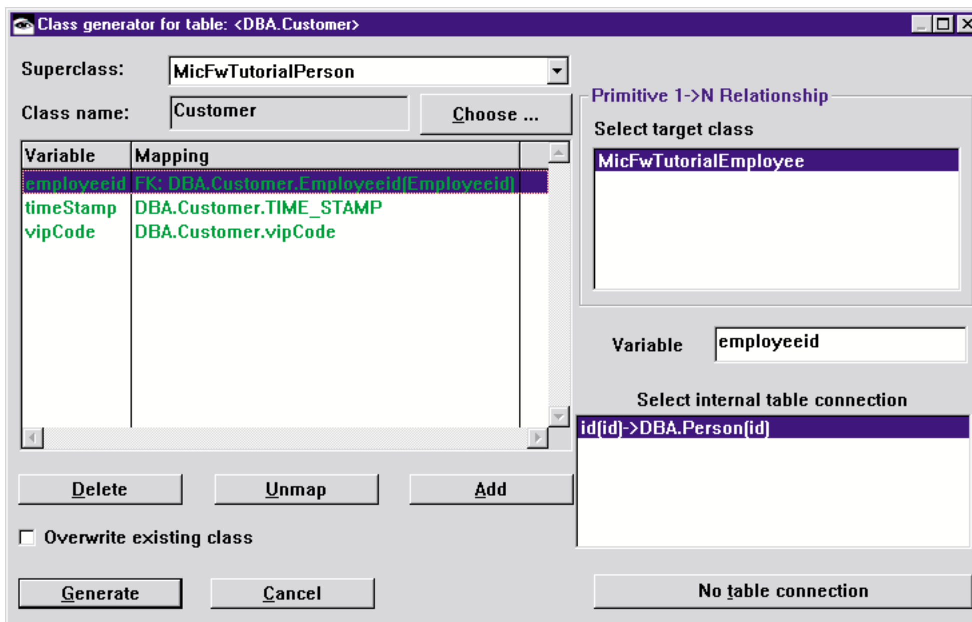


Figure 71: Mapped Class Generator

10.3. The POM Generator Tool

10.3.1. POM generator

The POM generator is used to generate a database schema and the mapping to this schema from existing persistent object classes. All included classes are subclasses of *MicFwPersistentObject* and all persistent instance variables have to be typed (base type or relationship). The POM generator will only include classes that fulfill one of the following prerequisites:

- classes which have at least one key variable.
- classes which have no key variables and are the target of a primitive to-1 relationship

The POM generator tool can be opened using menu item *micFrameworks→Persistence Tools→"Object Net → POM Generator"*

The opened dialog window allows you to specify all the parameters required for a POM generation process.

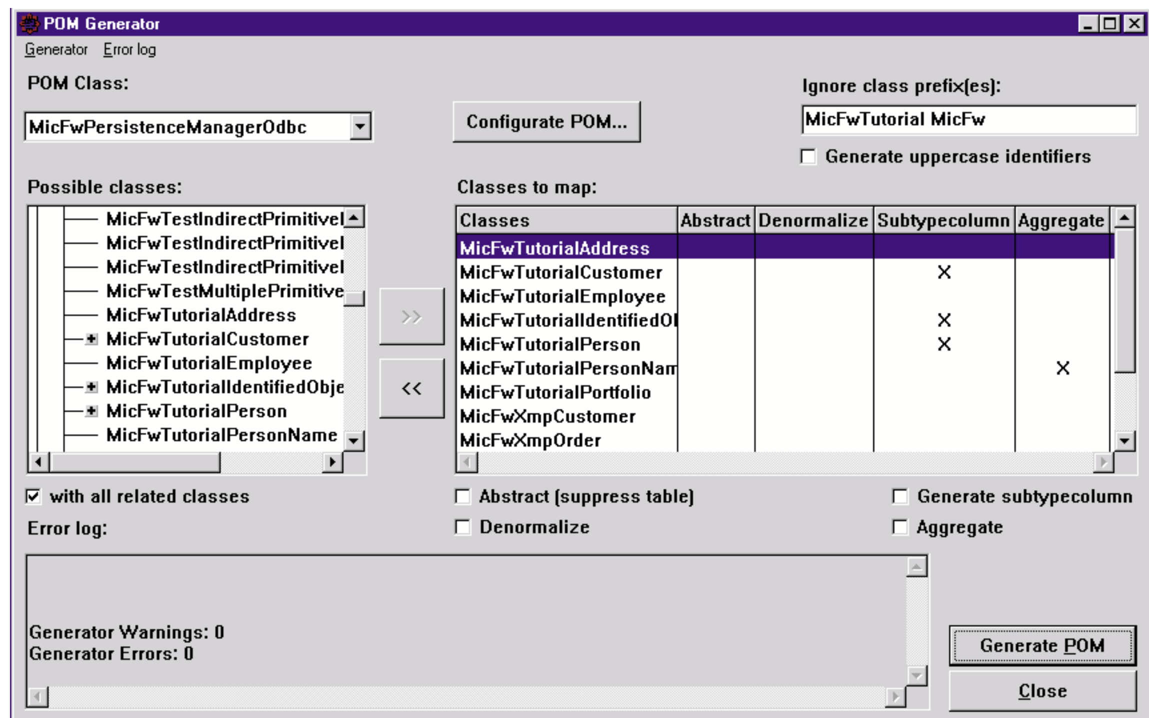


Figure 72: POM generator dialog window

10.3.1.1. POM Class

Specifies the class of the POM instance that will be generated.

10.3.1.2. POM Configuration

Opens the POM configuration window that allows you to specify the database description, commit handler, statement executor, etc.

Note: You should use the POM configuration window to at least specify the proper "Database Description" and "StatementExecutor" for the target database system for which you want a table schema to be generated. Otherwise you might get API errors during SQL execution because the POM generator has created data types that the target database does not support.

10.3.1.3. Classes to map

Specifies classes for which a POM will be generated.

10.3.1.4. With all related classes

Includes all classes related to the selected classes in the "Possible classes" listbox. When clicking on the ">>" button, all subclasses, superclasses and classes referenced by relationships of the selected classes will also be moved to the "Classes to map" listbox.

10.3.1.5. Ignore class prefix

This option is used to suppress class prefixes for the generated table names. More than one prefix can be specified by separating the prefix strings with blanks (e.g. '***MicFwTutorial MicFwXmp***').

10.3.1.6. Generate uppercase identifiers

Generate all identifiers (i.e. table names and column names) in upper case. This option should be used if the target database schema is not case sensitive.

10.3.1.7. Abstract (Suppress Table)

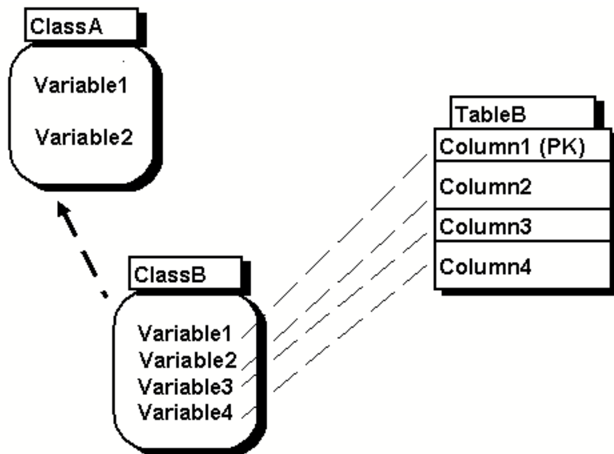


Figure 73: Suppress table

No tables are generated for the classes selected in the “Classes to map” listbox. If the selected classes in the object model to be mapped possess subclasses, the attributes of the selected classes are copied with redundance into the subclass tables.

Note: A class cannot be “suppressed” if it has persistent relationships to other classes.

10.3.1.8. Denormalize

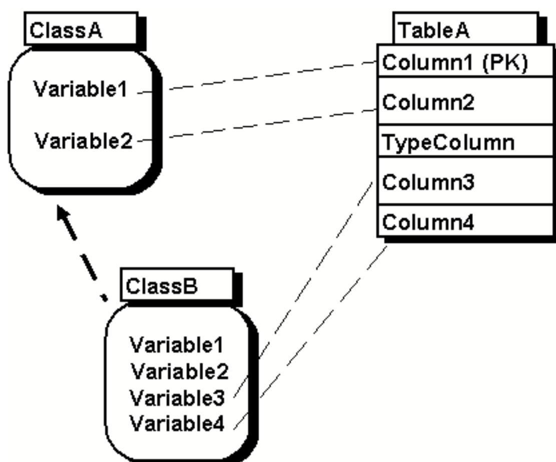


Figure 74: Denormalize table

No tables are generated for the classes selected in the “Classes to map” listbox. All attributes of the selected classes are transferred to the tables of the relevant super-classes.

Note: A class must satisfy the following conditions in order that it can be denormalized:

- the class must have a super-class in the object net which is to be mapped
- the class must not have any ‘&Key’8 variables of its own.
- the class must not have any existential to-1 relationships to other classes in the object net

10.3.1.9. Generate subTypeColumn

Generates a *subTypeColumn* for all classes that are marked in the *Classes to map* list box. The subtype values for subclasses are set appropriately.

Note: The subType column is generated for all marked classes, irrespective of whether the respective class contains a subclass.

10.3.1.10. Aggregate

Aggregates the classes marked in the *Classes to map* list box to their target classes.

10.3.1.11. Generate POM

Start the generation process. A STOPF browser will be opened for the newly generated POM. Errors and warnings will be displayed in the logging area in the lower part of the window.

10.3.1.12. Generator Menu

Figure 75: Menu Generator

- **New –** Deletes all settings and creates a new POM generator.
- **Choose** – Loads a POM generator that is stored in the image.
- **Import From File –** Loads a POM generator that is stored in a **.PGN* file.
- **Delete** – Removes the current POM generator from the image.
- **Find class –** Set marking in the *Possible classes* list box to the specified class.
- **Configure POM –** configure the generated POM (see 1.1.2 POM Configuration)
- **Generate POM -** (see 1.1.10 Generate POM)
- **Store globally as –** Stores the POM generator settings in the current image. Settings stored in this way can be retrieved using the *Choose* menu item. **Note:** Remember to save the image, otherwise the stored settings will be irretrievably lost.
- **Export To File –** Stores the POM generator settings in executable Smalltalk code in a **.PGN file.* Settings stored in this way can be retrieved using the *Import from file* menu item.
- **Close –** Closes the POM generator dialog box.

10.3.1.13. Error Log Menu

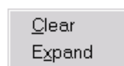


Figure 76: Menu Error Log

- **Clear** – Deletes the contents of the *Errorlog* list box.
- **Expand** – Opens a workspace window containing the contents of the *Errorlog* list box.

10.3.1.14. Errors

No.	Error message	Description
E1	<classname> has no persistent key variable.	You have attempted to load an object net into a class that does not have any persistent ”Key” variables.
E2	Cannot generate mapping for primitive n-1 relationship <columnname> in class <class-name>	You have attempted to generate a primitive n-1 relationship.
E3	Table for <classname> cannot be denormal-ized, because <columnname> is a mandatory attribute.	You have attempted to denormalize a table that has its own mandatory attributes.
E4	Table for <classname> cannot be generated.	Check whether all the conditions for table generation have been satisfied.
E5	Table for <classname> cannot be sup-pressed.	You have attempted to denormalize its sub-class

No.	Error message	Description
E6	Table for <classname> cannot be denormalized, because no superclass table exists.	You have attempted to denormalize a table for which there is no superclass. Or the superclass table has not been created for some reason (e.g. superclass table was suppressed).
E7	Recursive key relationship detected:	Check the key variables of both sides. Only one side of the relationship may contain a key variable.
E8	Cannot denormalize <classname> because <columnname> is a key attribute.	You have attempted to denormalize a table that contains a key variable.
E9	<classname>>><columnname> is a key relationship, but target class <classname> is not selected.	Check whether the target class has been loaded into the object net.
E10	Cannot aggregate <aRelationship> because <columnname> in <classname>has a relationship definition.	You have attempted to aggregate a class that contains a relationship.
E11	Cannot mapped <columnname> from table to class <classname> because class <classname> has no table.	Check whether the variables have different structures (e.g.: Type, Key, ...).
E12	Cannot generate foreign key from <classname> to <classname> because the primary keys do not match.	Check the primary keys of both classes. You have probably overwritten the inherited primary key.
E13	Table for <classname> cannot be denormalized, because no persistentKeyVariable exists.	Check the class that you want to denormalize. The class must contain a persistent key variable
E14	<columnname> in Class <classname> has incompatible type.	Check the type of the variable.
E15	Cannot aggregate <classname> in class <classname> because 1To1 relationship is not yet supported.	You have attempted to replace a 1:1 relationship with an aggregation. This function is not yet supported.
E16	Table for <classname> cannot be Aggregated, because <classname> has subclass in object net.	You have attempted to aggregate a class that contains subclasses.
E17	Table of <classname> has no Primarykey column	Check whether the class has a primary key.

Table 18: List of error messages from the POM generator

10.3.1.15. Warnings

No.	Warning message	Description
W1	the class <classname> must be aggregated in class <classname>	You have attempted to generate a table for an aggregated class.
W2	Column <columnname> in table <tablename> is incompatible with column <columnname> in table <tablename>	You have modified the inherited variables in the class and made them incompatible.

Table 19: List of warning messages from the POM generator

10.4. Interactive SQL

10.4.1. Overview

The Interactive SQL tool gives you the ability to interactively execute any direct SQL statements from within your Smalltalk environment. This tool uses the direct SQL interface of a POM to send the statements to the respective API (the direct SQL interface API is the same for all subclasses of ***MicFwPersistenceM-anagerRdb***.You can use this tool for testing or executing the DDL (Data Description Language) generated using the STOPF schema export facility (refer to for a description of the schema export feature).

You can open the "Interactive SQL" tool by selecting "Persistence Tools → Interactive SQL" from the "mic-Frameworks" menu in your Smalltalk Transcript window or by evaluating the following expression:

```
MicFwISQLBrowser openBrowser
```

10.4.2. Usage

Before you can execute any SQL statement, you need to select a POM using menu item *"Man-ager→Choose..."*. After you have selected a POM from the list, the tool uses this POM's direct SQL interface to communicate with the underlying API. The name of the selected POM appears in the title bar of the window.

Note: The Interactive SQL Browser can work with tool-POMs as well as with RT-POMs!

The selected POM must be connected to a database. If it is not already connected, you can establish a database connection using menu item "Manager→Connect...". To disconnect the POM, use menu item "Manager→Disconnect". The title bar of the window indicates whether the selected POM is connected.

10.4.2.1. The browser window

The window of the "Interactive SQL" browser has an upper and a lower text window:

10.4.2.1.1. The upper text window (statement window)

Enter the statements you want to execute in the upper text window. You can enter any valid SQL statement or Smalltalk expression here. To execute the expression, use the items in the "Statement" menu. For more than one SQL statement, separate each SQL statement with ;

You can

- load any ASCII file (containing statements) into this window using menu item "File→Open...".
- save the contents of this window using menu item "File→Save" or "File→Save As...".

10.4.2.1.2. The lower text window (result window)

This text window is used to display the results of SQL statement executions, e.g. the results from a SELECT query.

10.4.2.2. The "Statement" menu

10.4.2.2.1. Execute once

Choose this item if you want to execute the SQL statement(s) in the statement window without expecting any results (e.g. for UPDATE, INSERT, DELETE or CREATE TABLE statements, etc.). The statement will be executed and the database cursor for this statement is closed immediately.

10.4.2.2.2. Execute for fetch

Choose this item if you want to execute the SQL statement in the statement window and want to retrieve results from the statement (e.g. for SELECT queries). The statement will be executed immediately and the database cursor is opened for fetching the answer set rows. The result window displays the column names of the expected query result.

10.4.2.2.3. Execute and fetch all

This item executes the SQL (SELECT) statement in the statement window and fetches all rows of the answer set immediately. The result is displayed in the result window and the cursor will be closed after all rows have been fetched.

10.4.2.2.4. Fetch next

This item is enabled when you previously selected "Execute for fetch", and thus a database cursor is open to retrieve rows. The editor fetches the next row from the answer set and displays it in the result window.

10.4.2.2.5. Fetch all

This item is enabled when a database cursor is open. Fetch all rows from the current database cursor posi-

tion to the end of the answer set and display them in the result window.

10.4.2.2.6. Commit

Perform a transaction commit on the database.

10.4.2.2.7. Rollback

Perform a transaction rollback on the database.

10.4.2.2.8. Close cursor

When a database cursor is open, you can close it using this menu item. Do not forget to close your database cursors, because the SQL interface of the Framework will not do this for you automatically.

10.4.2.2.9. Clear Statement

Clear the contents of the statement window.

10.4.2.2.10. Clear results

Clear the contents of the result window.

10.4.2.3. The "Smalltalk" menu

10.4.2.3.1. Execute

Use this menu item if the statement window contains proper Smalltalk expressions instead of SQL. This lets you perform any Smalltalk statement within the "Interactive SQL" tool.

The receiver of the Smalltalk expression (‘self’) is the selected POM.

Try the following example. Enter

```
MicFwISQLBrowser openBrowser.
```

into the statement window and select "Execute" from the "Smalltalk" menu. A second *Interactive SQL Browser* opens.

10.4.2.3.2. Inspect

If you want to inspect an result of a smalltalk statement, so you can inspect it with this menu item.

An example: Type the following statement

```
self loadAllInstancesOf: MyCustomerClass.
```

into the statement window and select "Inspect" from the "Smalltalk" menu. A Smalltalk inspector opens.

Remember: A POM logs all SQL statements that are executed in the Framework logger. This also applies to SQL statements executed using the "Interactive SQL" tool.

Appendix A

API

The following is a list of API methods for the Persistence Framework.

MicFwPcmProcessClassDescription>>
addBaseConnectionDescription: aBaseClassName

Create a new base connection and add it to the receivers connection descriptions.

MicFwPcmProcessClassDescription>>
addChildProcessConnectionDescription: aConnDesc

Add a new connection description.

MicFwPcmProcessClassDescription>>
allBaseConnectionDescriptions

Answer a collection with all involved base entries from the receiver.

MicFwPcmProcessClassesNet>>
allEnvironmentNames

MicFwPcmProcessClassesNet>>
allEnvironments

MicFwObject class>>
allInstVarSymbols

Answer a collection with all instance variable names as symbols.

MicFwPcmProcessClassDescription>>
allProcessConnectionDescriptions

Answer a collection with all process entries from the receiver.

MicMlsLCTime>>
amString

MicMlsLCTime>>
amString: aString

MicFwNRelationship>>
any

Answer a target object or <nil>. If there are more than one target objects in the collection, it is not ensured, that multiple calls of this method would return the same target object.

MicFwEnvironmentConnector>>
asListEntry

MicFwTypeDescription>>
assertValue: aValue

Attempt to convert <aValue> to the type described by the receiver. Answer the new Object. Signal an Exception if the conversion is not possible.

MicMlsLCMessages>>
at: aKey put: aValue

Accessing the internal structure.

MicMlsLocaleCategory>>
at: aKey put: aValue

Accessing the internal structure.

MicFwStreamWriter>>
atEnd

Answer a Boolean which is true if the receiver write stream cannot access any more objects, and false otherwise.

MicFwObject>>
attributeNamed: instVarSymbol

Answer the value of an instance variable by getting the instvar index from the class and accessing it via #instVarAt:.

MicFwObject>>

attributeNamed: instVarSymbol put: aValue

Perform a write to an instance variable by getting the instvar index from the class and writing to it via #instVarAt:.

MicFwPcmProcessClassDescription>>
baseConnectionDescriptions

Answer a collection with involved base entries except default base.

MicFwPcmProcessClassesNet>>
canChangeDestinationEnvironment

MicFwPcmProcessClassesNet>>
canChangeSourceEnvironment

MicFwTypeDescription>>
canConvert: aValue

Answer true if <aValue> can be converted to the type described by the receiver, or false otherwise.

MicFwPcmProcessClassesNet>>
changeDestinationEnvironmentTo: anEnvironment

MicFwPcmProcessClassDescription>>
childProcessConnectionDescriptions

Answer a collection with all child process entries from the receiver.

MicFwPcmProcessClassDescription>>
childTransactionDependentSubclasses

Answer a collection of all subclasses which are dependent on the receivers child transaction behavior.

MicFwStreamWriter>>
close

Close any underlying file associated with the receiver's write stream.

MicFwCommander class>>
commanderName

Answer the receiver's name, which is used as an identification to be independent of class references.

MicMlsLocaleContext>>
connectedLocales

Returns the currently connected locales.

MicMlsLocaleContext>>
connectLocaleFor: aLC

Connect a locale entity of the given locale category and a name which is the locale category class name.

MicMlsLocaleContext>>
connectLocaleFor: aLC named: aLocaleName

Connect a locale entity of a given locale category with a locale name aLocaleName.

All data will be loaded into cache.

MicMlsLocaleContext>>
connectSubcontext: aLocaleContext

Connect aLocaleContext as sub context. Localize aLocaleContext and it's locale objects and all its subcontexts to the locale of self.

MicMlsLocaleContext>>
connectSupercontext: aLocaleContext

Connect aLocaleContext as super context. Localize self to the locale of aLocaleContext.

MicMlsLocaleContextDescription>>
connectToMasterContext

MicMlsLocaleContextDescription>>
connectToMasterContext: aValue

MicFwStreamWriter>>
contents

Answer a Collection which is a copy collection that the receiver's write stream is streaming over, truncated to the current position reference.

MicMlsLocaleContext>>
contextDescription

Return the context description.

MicMlsLocaleContextDescription>>
contextHierarchy

MicMlsLocaleContextDescription>>
contextHierarchy: aCollection

MicFwTypeDescription>>
converterClass

Answer the class used for type conversion.

MicFwStreamWriter>>
copyFrom: firstIndex to: lastIndex

Answer the subcollection of the collection over which the receiver's write stream is streaming, from @firstIndex to @lastIndex.

MicFwStreamWriter>>
cr

Store the cr (carriage return) character and the current indent as the next objects in the receiver's write stream. Answer self. Set the receiver's position reference to be immediately after the cr character.

MicFwEnvironmentConnector>>
createDomainObjectClasses

MicFwJavaConnector>>
createDomainObjectClasses

MicFwEnvironmentConnector>>
createMlsContextClasses

MicFwJavaConnector>>
createMlsContextClasses

MicFwEnvironmentConnector>>
createProcessClassesDictionary

MicFwJavaConnector>>
createProcessClassesDictionary

MicFwEnvironmentConnector>>
createViewClasses

MicFwJavaConnector>>
createViewClasses

MicFwEnvironmentConnector>>
createViewPortClasses

MicFwJavaConnector>>
createViewPortClasses

MicFwTransactionContext class>>
current
Answer the current transaction context or nil if none active. This is an alternative to MicFwTransactionManager global currentContext.

MicFwTransactionManager>>
currentContext
Return the active context.

MicFwTransactionManager class>>
currentContext
Return active context (or nil).

MicMlsLCDate>>
dayAndMonthStrings

MicMlsLCDate>>
dayAndMonthStrings: aMicMlsLocaleMessages

MicMlsLCDate>>
dayOfWeek: aDate

MicFwStreamWriter>>
decreaseIndent
Decrease the current indent by 1.

MicFwStreamWriter>>
decreaseIndent: anInteger
Decrease the current indent by @anInteger.

MicFwStreamWriter>>
decreaseIndentWhile: aBlock
Decrease the receiver's indent by 1 while executing the zero argument block @aBlock.Answer the result value of @aBlock.

MicFwPcmProcessClassDescription>>
defaultBaseConnectionDescription
Answer the involved default base entry.

MicFwPcmProcessClassDescription>>
defaultBaseConnectionDescriptions
Answer a collection with involved default base entry.

MicFwViewPort>>
defaultChildPaneConnectionModeValueOf: dispatcherAspect
Answer the default late model access value for the <dispatcherAspect>.

MicMlsLocaleCategory class>>
defaultKeyStringsLocaleName

MicMlsLocaleContextDescription>>
defaultLocale

MicMlsLocaleContextDescription>>
defaultLocale: aValue

MicFwViewPortRequestBroker class>>
defaultViewPortSymbol
Return the symbol for the default viewport class.

MicFwPcmTransactionBehavior class>>
defaultWith: aDescribedClassName
Create a new MicFwPcmTransactionBehavior from original transaction behavior.

MicFwEnvironmentProcess>>
defineEnvironmentFor: aBrowser

MicMlsLocaleContext>>
delegateDelocalizationOf: aStringObject to: aLC withNameExtension: aLocaleNameExtension withKey: aKeyString withTargetClass: aClass

Delegates delocalization of aString to an object of target class aClass.

While going up the context hierarchie of self, each locale entity with a name which ends on aLocaleName-Extension is searched through.

For each locale category which uses an additional Object (e.g. LCNumeric), aKeyString defines which format string should be used to delocalize the String . When the given key string is not found the target class decides which default keys should be used.

Return nil if delocalization failed.

MicMlsLocaleContext>>
delegateDelocalizationOf: aStringObject to: aLC withNameExtension: aLocaleNameExtension withKey: aKeyString withTargetClass: aClass allowingErrors: aBoolean

Delegates delocalization of aString to an object of target class aClass.

While going up the context hierarchie of self, each locale entity with a name which ends on aLocaleName-Extension is searched through.

For each locale category which uses an additional Object (e.g. LCNumeric), aKeyString defines which format string should be used to delocalize the String . When the given key string is not found the target class decides which default keys should be used.

Return nil if delocalization failed.

MicMlsLocaleContext>>
delegateDelocalizationOf: aStringObject toTargetClass: aClass

Delegates delocalization of aString to an object of target class aClass.

Return nil if delocalization failed.

MicMlsLocaleContext>>
delegateDelocalizationOf: aStringObject toTargetClass: aClass allowingErrors: aBoolean

Delegates delocalization of aString to an object of target class aClass.

Return nil if delocalization failed.

MicMlsLocaleContext>>
delegateDelocalizationOf: aStringObject withKey: aKeyString toTargetClass: aClass

Public- Delegates delocalization of aString to an object of target class aClass using aKeyString. Return nil if delocalization failed.

MicMlsLocaleContext>>
delegateDelocalizationOf: aStringObject withKey: aKeyString toTargetClass: aClass allowing-Errors: aBoolean

Delegates delocalization of aString to an object of target class aClass using aKeyString.

Return nil if delocalization failed.

MicMlsLocaleContext>>
delegateLocalizationOf: anObject

Delegates localization of anObject using a locale category aLC - While going up the context hierarchie of self, the first locale entity of the given locale category of each context is searched through. Trigger an Mic-MlsAccessingKeyError if localization failed.

Answer nil if an illegal object was given.

MicMlsLocaleContext>>
delegateLocalizationOf: anObject to: aLC

Delegates localization of anObject using a locale category aLC.

While going up the context hierarchie of self, the first locale entity of the given locale category of each context is searched through.

Trigger an MicMlsAccessingKeyError if localization failed.

MicMlsLocaleContext>>
delegateLocalizationOf: anObject to: aLC named: aLocaleName

Delegates localization of anObject using a locale category aLC and a name of a locale entity aLocale-

Name While going up the context hierarchie of self, each locale entity which has the given name and locale category is searched through. Trigger an MicMlsAccessingKeyError if localization failed.

MicMlsLocaleContext>>
delegateLocalizationOf: anObject to: aLC named: aLocaleName withKey: aKeyString

Delegates localization of anObject using a locale category aLC, a name of a locale entity aLocaleName and aKeyString which is the key of a format string. While going up the context hierarchie of self, each locale entity which has the given name and locale category is searched through. For each locale category which uses an additional Object (e.g. LCNumeric), aKeyString defines which format string should be used to localize the String . When the given key string is not found, the class of anObject decides which default keys should be used. Trigger an MicMlsAccessingKeyError if localization failed.

MicMlsLocaleContext>>
delegateLocalizationOf: anObject to: aLC withNameExtension: aLocaleNameExtension

Delegates localization of anObject using a locale category aLC and the ending of a name of a locale entity aLocaleNameExtension While going up the context hierarchie of self, each locale entity of the given locale category which locale name ends on aLocaleNameExtension is searched through. Trigger an MicMlsAccessingKeyError if localization failed.

MicMlsLocaleContext>>
delegateLocalizationOf: anObject to: aLC withNameExtension: aLocaleNameExtension withKey: aKeyString

Delegates localization of anObject using a locale category aLC, the and the ending of a name of a locale entity aLocaleNameExtensio and aKeyString which is the key of a format string. While going up the context hierarchie of self, each locale entity of the given locale category which name ends on aLocaleNameExtension is searched through. For each locale category which uses an additional Object (e.g. LCNumeric), aKeyString defines which format string should be used to localize the String . When the given key string is not found, the class of anObject decides which default keys should be used. Trigger an MicMlsAccessingKeyError if localization failed.

MicMlsLocaleContext>>
delegateLocalizationOf: anObject to: aTargetLC withNameExtension: '' withKey: aKeyString

MicMlsLocaleContext>>
delegateLocalizationOf: anObject withKey: aKeyString

PublicDelegates localization of anObject using anObjects locale category aLC and an empty name extension While going up the context hierarchie of self, the first locale entity of the given locale category in each context is searched through. Trigger an MicMlsAccessingKeyError if localization failed Return nil if anObject can't be converted.

MicFw1Relationship>>
delete

Delete the relationship maintained by the receiver. Also delete the inverse references.

MicMlsLCKeys>>
deLocalize: anObject

MicMlsLCStrings>>
deLocalize: anObjectAsString targetClass: anObjectClass withKey: aKeyString

MicMlsLocaleCategory>>
deLocalize: anObjectAsString targetClass: targetClass withKey: aKeyString allowingErrors: aBoolean

Delocalize aString using the key aKeyString the result is an error or an object of class targetClass

MicMlsLocaleCategory>>
deLocalize: aString targetClass: targetClass

Delocalize aString using the 'default' key the result is an error or an object of class targetClass

MicMlsLocaleContext>>
disconnectLocale: aLocaleEntity

Diconnect a locale entity from self

MicMlsLocaleContext>>
disconnectLocaleFor: aLC

Disconnect a locale entity of the given locale category and a name which is the locale category class name

MicMlsLocaleContext>>
disconnectLocaleFor: aLC named: aLocaleName

Disconnect a locale entity of a given locale category with a locale name aLocaleName

MicMlsLocaleContext>>
disconnectSubcontext: aLocaleContext

Disconnect aLocaleContext as sub context

MicMlsLocaleContext>>
disconnectSupercontext

Disconnect self from its super context

MicFwStreamWriter>>
do: aBlock

Evaluate the one argument block, @aBlock for each of the remaining elements accessible by the receiver's write stream.

MicFwEnvironmentConnector>>
equals: anEnvironment

MicFwObject class>>
extendedDescription

Answer the extended description.

MicIsInterface class>>
extends

Answer a collection of interfaces which the receiver extends.

MicFwStreamWriter>>
flush

Write all elements of the receiver's write stream to any underlying file.

MicMlsLCDate class>>
formatSettingVariablesDescription

MicMlsLCMonetary class>>
formatSettingVariablesDescription

MicMlsLCNumeric class>>
formatSettingVariablesDescription

MicMlsLCTime class>>
formatSettingVariablesDescription

MicMlsLCDate class>>
formatVarDescriptor

MicMlsLCMessages class>>
formatVarDescriptor

MicMlsLCMonetary class>>
formatVarDescriptor

MicMlsLCNumeric class>>
formatVarDescriptor

MicMlsLCStrings class>>
formatVarDescriptor

MicMlsLCTime class>>
formatVarDescriptor

MicMlsLocaleCategory class>>
formatVarDescriptor

MicMlsLCTime class>>
formatVariablesDescription

MicMlsLCDate class>>
formatVariablesDescription

MicMlsLCMonetary class>>
formatVariablesDescription

MicMlsLCNumeric class>>
formatVariablesDescription

MicMlsLocaleCategory class>>
formatVariablesDescription

Returns the description of the used format variables.

MicMlsLCTime>>
fullDay

MicMlsLCTime>>
fullDay: aBool

MicMlsLCMonetary>>
fullName

MicMlsLCMonetary>>
fullName: aString

MicFwEnvironmentConnector>>
generateDescriptionFor: aProcessClassDesc

MicFwJavaConnector>>
generateDescriptionsFor: aCollection

MicFwEnvironmentConnector>>
generateDomainProcessDescriptionsFor: aCollection

MicFwPersistenceManagerOdbc>>
getAllInstancesOf: aDO

Returns a Collection of all instances of the DO found in the database.

MicFwPcmProcessClassDescription>>
getChildTransactionBehavior

Answer the receivers currently used child transaction behavior.

MicFwEnvironmentConnector>>
getClassCollections

MicMlsLocaleEntitySearcher>>
getCurrentLocale

MicFwEnvironmentConnector>>
getDomainProcessClasses

MicFwPcmProcessClassDescription>>
getMainTransactionBehavior

Answer the receivers currently used main transaction behavior.

MicFwEnvironmentConnector>>
getRefreshedClassCollections

MicFwEnvironmentConnector>>
getRefreshedDomainProcessClasses

MicFw1Relationship>>
getTarget

Returns the target object of the ->1 relationship.

MicFwNRelationship>>
getTarget

Returns a Collection of target objects referenced by the ->N relationship.

MicFwRelationship>>
getTarget

Returns the target object (the actual object or a collection, depending on the type of relationship) of the relationship.

MicFwTr1Relationship>>
getTarget

Returns the transacted target object of the ->1 relationship.

MicFw1Relationship>>
getTargetNoTr

For an non-transacted relationship: Returns the object referenced by the ->1 relationship. For a transacted relationship: Returns the variable value (not the transacted object).

MicFwNRelationship>>
getTargetNoTr

For an non-transacted relationship: Returns the object version of the Collection of objects referenced by the ->N relationship. For a transacted relationship: Returns the variable value of the Collection (not the transacted object).

MicFwRelationship>>
getTargetNoTr

For an non-transacted relationship: Returns the object referenced by the relationship. For a transacted relationship: Returns the variable value (not the transacted object).

MicFwTransactionManager class>>
global

Return single instance of receiver.

MicFwPcmProcessClassDescription>>
hasChildTransactionDependentSubclasses

Answer true if the receiver has subclasses which are dependent on his child transaction behavior.

MicFwTransactionManager>>
hasCurrentContext

Return True if there is a active context.

MicFwPcmProcessClassDescription>>
hasDefaultBaseDependentSubclasses

Answer true if the receiver has subclasses which are dependent on his child transaction behavior.

MicFwPcmProcessClassDescription>>
hasMainTransactionDependentSubclasses

Answer true if the receiver has subclasses which are dependent on his main transaction behavior.

MicFwTransactionManager>>
hasRunningTransaction

Return True if the active context has a TrLevel1.

MicMlsLocaleContext>>
hasSubcontext

Returns if the context has a subcontext or not.

MicMlsLocaleContext>>
hasSupercontext

Returns if the context has a super context or not.

MicMlsLocaleContext>>
includesLocaleFor: aLC

Answer if a locale entity of a locale category is connected to this context.

MicMlsLocaleContext>>
includesLocaleFor: aLC named: aLocaleName

Answer if a locale entity named aLocaleName of a locale category aLC is connected to this context.

MicFwStreamWriter>>
increaseIndent

Increase the current indent by 1.

MicFwStreamWriter>>
increaseIndent: anInteger

Increase the current indent by @anInteger.

MicFwStreamWriter>>
increaseIndentWhile: aBlock

Increase the receiver's indent by 1 while executing the zero argument block @aBlock. Answer the result value of @aBlock.

MicFwStreamWriter>>
indent

Answer the receiver's current indent.

MicFwStreamWriter>>
indentCharacter

Answer the receiver's indent character.

MicFwStreamWriter>>
indentCharacter: aCharacter

Set the receiver's ident character to @aCharacter.

MicFwObject class>>
initializeAll

Iinitialize the extended description of the receiver and all subclasses.

MicFwPcmProcessClassesNet>>
initializeEnvironment

MicFwDomainObject class>>
initializeValidation

Class method that sends all validateWrite:using: messages for all instance variables of the class that should be validated.

MicFwPcmProcessClassDescription>>
initializeWith: aClassName with: aSuperclass with: aSubclassCollection with: anInstVarCollection with: aDefaultInvolvedBaseName with: aDefaultViewState with: aDefaultViewClass with: aConnectionDescriptionCollection with: aMainTransactionHandling with: aMainTransactionBehavior with: aChildTransactionHandling with: aChildTransactionBehavior

Initialize receiver with values from original process.

MicFwRelationship>>
is1ToN

Answer whether the receiver describes a 1-to-N relationship.

MicFwTransactionContext>>

isActive

Answers True if receiver is the single active context.

MicFwEnvironmentConnector>>
isActive

MicFwJavaConnector>>
isActive

MicFwPersistenceManagerOdbc>>
isConnected

Returns true if the POM is connected to the database.

MicCwPanel>>
isDatePanel

Returns if a date panel.

MicMlsDomainLocaleContext class>>
isDummyContext

Return if this context is a dummy context.

MicFwStreamWriter>>
isEmpty

Answer a Boolean which is true if the receiver's write stream can access any objects and false otherwise.

MicFwTransactionContext>>
isIsolated

Answers True if receiver mode is isolated (not uncommitedRead).

MicCwPanel>>
isLabelPanel

Return is a label panel.

MicMlsDomainLocaleContext class>>
isMasterContext

Return if the class is a master context.

MicFwRelationship>>
isMicFwRelationship

Answer whether the receiver is a relationship object.

MicMlsLocaleCategory>>
isMlsDataTransacted

Answer if MLS data of this locale entity should be transacted.

MicMlsLocaleCategory>>
isMlsDataTransacted: aBoolean

Decide if MLS data of this locale entity should be transacted.

MicFwRelationship>>
isNToM

Answer whether the receiver describes a N-to-M relationship.

MicFwTransactionContext>>
isolate

Set the mode of the receiver to #isolate. No uncommited changes from other transaction contexts will be visible through this context.

MicFwRelationship>>
isPrimitive

Answer whether the receiver is a primitive (one-sided) relationship object.

MicMlsLocaleContextDescription>>
isRealContext

MicMlsLocaleContextDescription>>
isRealContext: aValue

MicFwTransactionContext>>
isRunning
Answer whether the receiver is in an uncommited state.

MicCwPanel>>
isTimePanel
Return if a time panel.

MicFw1Relationship>>
isTo1
Answer whether the receiver describes a to-1 relationship.

MicFwRelationship>>
isTo1
Answer whether the receiver describes a to-1 relationship.

MicFwNRelationship>>
isToN
Answer whether the receiver describes a to-N relationship.

MicFwRelationship>>
isToN
Answer whether the receiver describes a to-N relationship.

MicFw1Relationship>>
isValid
Answer true if the receiver conforms to the declared cardinality constraints, false otherwise.

MicFwNRelationship>>
isValid
Answer true if the receiver conforms to the declared cardinality constraints, false otherwise.

MicFwRelationship>>
isValid
Answer true if the receiver conforms to the declared cardinality constraints, false otherwise.

MicMlsLocaleCategory class>>
keyStringsLocaleName

MicCwPanel>>
labelString
Return label string.

MicCwPanel>>
labelString: aString
Set the value in the widget.

MicFwEnvironmentConnector>>
language

MicFwJavaConnector>>
language

MicMlsLocaleCategory>>
language

MicMlsLocale>>
language: aLangString territory: aTerrString

MicMlsLocale class>>
language: aLangString territory: aTerrString

MicMlsLocaleCategory>>

language: aLanguage

MicFwStreamWriter>>
lineDelimiter
Return the receiver's write stream line delimiter.

MicFwStreamWriter>>
lineDelimiter: delimiter
Set the receiver's write stream line delimiter to be delimiter, and answer the receiver.

MicMlsLocaleCategory>>
locale
Returns the current locale.

MicMlsLCDate>>
locale: aLocaleDescription

MicMlsLocaleCategory>>
locale: aLocaleDescription
Set the current locale.

MicMlsLocaleCategory>>
locale: aLocaleDescription named: aLocaleName

MicMlsLocaleCategory>>
localeCategory

MicMlsLocaleCategory>>
localeCategory: aLC

MicMlsLocaleContextDescription>>
localeContextClass

MicMlsLocaleContextDescription>>
localeContextClass: aContextClass

MicMlsLocaleContextDescription>>
localeEntityDescriptionsToBeConnected

MicMlsLocaleContextDescription>>
localeEntityDescriptionsToBeConnected: aValue

MicMlsLocaleCategory>>
localeName

MicMlsLocaleCategory>>
localeName: aString

MicFwViewPort>>
localize: aLocale
The user wants to switch the standard context hierarchy.

MicMlsLocaleContext>>
localize: aLocale
Set the locale of self to aLocale and set all connected locales and sub contexts to the new locale.

MicMlsLocaleCategory>>
localize: anObject
Localizes <anObject> for it's locale (#language, #territory) (-> COMMENT).

MicMlsLocaleCategory>>
localize: anObject withKey: aKeyString

Localizes anObject in self aKeyString is the key for the format string.

MicFwViewPort>>
localizeAllConnectionsTo: aLocale

Localize all mlsContexts within the actual connector to aLocale.

Object>>
log: aString

Logging Log an event into the #General category with #Info severity. See #log:severity:withString: for more info.

Object>>
log: categorySymbol severity: aSymbolOrInteger withString: aString

Logging Log an event for a category symbol.

aSymbolOrInteger can be a severity symbol (#Info, #Warning, #Error or #FatalError or an Integer (5 = #Primitive 4 = #Info, 3 = #Warning, 2 = #Error and 1 = #FatalError).

It's good style to use the symbols to allow later changes to the severity classes. The log is structured by two means, log severities and log categories. Severities categorize the importance of an event. There are four severities to use:

- #Primitive: A normal, technical primitive information event #Info Meaning a normal information event.
- #Warning: An unnormal situation that can be handled automatically.
- #Error: An error that fails the current operation.
- #FatalError: An error that makes the whole image fail Categories subdivide the log events into application contexts.

Each application should use it's own category name, usually the name of the (sub) application or a category name that speaks for itself (like #SQLExec). There is one default category named #General for general log events, that is used for the loggers own events and should be used only for events that are not application specific like events from the Exception Handling.

Object>>
log: categorySymbol withString: aString

Logging Log an event into a category symbol with #Info severity. See #log:severity:withString: for more info.

Object>>
logger

Logger Return the default logger of the image.

MicFwPcmProcessClassDescription>>
mainProcessConnectionDescriptions

Answer a collection with the main process connection description from the receiver.

MicFwPcmProcessClassDescription>>
mainTransactionDependentSubclasses

Answer a collection of all subclasses which are dependent on the receivers main transaction behavior.

MicMlsLocaleContextDescription>>
masterContextName

MicMlsLocaleContextDescription>>
masterContextName: aValue

MicMlsLCMessages>>
messages

MicMlsMonetaryObject class>>
micBaseFraction

MicMlsWeightWrapper class>>
micBaseFraction

Object>>
micFwDefaultPrimObjectViewPort

Return the class which should be used for the default primitive ViewPort.

MicMlsMonetaryObject>>
micMlsContentAfterConversionTo: aConversionString

Answer the number object which should represent the monetary value.

MicMlsWeightWrapper>>
micMlsContentAfterConversionTo: aConversionString

Answer the number object which should represent the monetary value.

Object>>
micMlsContentAfterConversionTo: aConversionString

Answer the converted object which should represent the object value.

MicMlsMonetaryObject class>>
micMultiplicationFraction

MicMlsWeightWrapper class>>
micMultiplicationFraction

MicMlsLCDate>>
millenniumSwitch

MicMlsLCDate>>
millenniumSwitch: anInt

MicMlsMonetaryObject class>>
mlsContent: aValue toBeConvertedFrom: aConversionString

Return the correct object for the model, using a value which must be converted to an object defined by aConversionString.

MicMlsWeightWrapper class>>
mlsContent: aValue toBeConvertedFrom: aConversionString

Return the correct object for the model, using a value which must be converted to an object defined by aConversionString.

Object class>>
mlsContent: aValue toBeConvertedFrom: aConversionString

Return the correct object for the model, using a value which must be converted to an object defined by aConversionString.

MicFwPcmProcessClassDescription>>
mlsContext

MicFwViewPort>>
mlsContext

Return the mlsContext of the connection this dispatcher belongs to.

MicFwPcmProcessClassDescription>>
mlsContext: aContext

MicMlsMonetaryObject class>>
mlsDefaultKeys

MicFwPcmProcessClassDescription>>
mlsState

MicFwPcmProcessClassDescription>>
mlsState: aSymbol

MicFwPcmProcessClassDescription>>
mlsSupercontext

MicFwPcmProcessClassDescription>>
mlsSupercontext: aContext

MicFwPcmProcessClassesNet>>
modifiedClassDescriptions

MicMlsLCDate>>
monthName: aNumber

MicMlsLCDate>>
monthNames

MicMlsLCDate>>
monthNumber: aString

MicFwPcmProcessClassDescription>>
name

MicFwPersistenceManagerOdbc>>
named: aString

Selects a POM (with name aString) as the persistence manager.

MicFwEnvironmentConnector>>
needPath

MicFwJavaConnector>>
needPath

MicFwObject class>>
new

Answer an initialized instance.

MicFwProcessesBrowserMain class>>
new

MicFwPcmConnectionDescription class>>
new: aConnectionType with: aProcessClassDescription with: aConnectionName with: aModelClass with: aMlsState with: aMlsContext with: aMlsSupercontext with: aViewPortClass with: aConnectionTransactionHandling with: aConnectionTransactionBehavior

MicFwPersistenceManagerOdbc>>
newPersistenceContext

Creates a new persistent context.

MicFwDomainObject>>
newPersistent

Creates a persistent instance of the DO.

MicFwTransactionContext>>
newTransactionContext

Create new context as child of receiver.

MicFwTransactionManager>>
newTransactionContext

Create a new <code>TransactionContext</code> with this manager as its owner and manager. This con-

text isn't activated automatically but can be activated by sending #beginTransaction to the context.

MicFwPcmProcessClassDescription class>>

newWith: aClassName with: aSuperclass with: aSubclassCollection with: anInstVarCollection with: aDefaultInvolvedBaseName with: aDefaultViewState with: aDefaultViewClass with: aConnectionDescriptionCollection with: aMainTransactionHandling with: aMainTransactionBehavior with: aChildTransactionHandling with: aChildTransactionBehavior

MicFwPcmTransactionBehavior class>>

newWith: aDescribedClassName with: anIsolateState with: aBeginTransactionState with: aUsePersistenceContextState with: aUseParentContextState with: aUseHierarchicalContextState

Initialize receiver with values from original transaction behavior.

MicFwPcmConnectionDescription class>>

newWith: aProcessClassDescription with: aConnectionName with: aModelClass with: aMlsState with: aMlsContext with: aMlsSupercontext with: aViewPortClass with: aConnectionTransactionHandling with: aConnectionTransactionBehavior

MicFwPcmConnectionDescription class>>

newWith: aProcessClassDescription with: aConnectionName with: aModelClass with: aMlsState with: aMlsContext with: aMlsSupercontext with: aViewPortClass with: aConnectionTransactionHandling with: aConnectionTransactionBehavior with: aTemplateDescription

MicMlsLCNumeric class>>

newWithSource: aSourceEnv withDestination: aDestinationEnv

MicFwStreamWriter>>

next

Answer an Object that is the next accessible by the receiver's write stream. Change the state of the receiver's write stream so that returned object is no longer accessible. Fail if the receiver's write stream is atEnd.

MicFwStreamWriter>>

next: anInteger

Answer a Collection containing the next @anInteger elements accessible by the receiver's write stream. Change the state of the receiver's write stream so that the next @anInteger elements are no longer accessible. If @anInteger < 1 then answer an empty Collection. Fail if @anInteger is not a kind of Integer Fail if there are less than @anInteger elements left accessible by the receiver.

MicFwStreamWriter>>

next: anInteger put: anObject

Store argument @anObject into the next @anInteger elements accessible to the receiver's write stream. Answer @anObject. Change the state of the receiver so that the anInteger elements of qanObject are not accessible. If anInteger < 1, then do not store any instances of @anObject. Fail if @anInteger is not a kind of Integer.

MicFwStreamWriter>>

nextLine

Answer the elements between the current position and the next lineDelimiter.

MicFwStreamWriter>>

nextMatchFor: anObject

Answer a Boolean which is true if the next element in the receiver's write stream is equivalent (=) to the argument @anObject and false otherwise. Change the state of the receiver's write stream so that the element that argument @anObject is compared against is inaccessible. Fail if there are no more objects to be accessed by the receiver.

MicFwStreamWriter>>

nextPut: anObject

Store the argument @anObject at the next position accessible to the receiver's write stream. Answer anObject. Change the state of the receiver so that the argument anObject is no longer accessible.

MicFwStreamWriter>>
nextPutAll: aCollection

Store each of the elements of @aCollection starting at the current position accessible to the receiver's write stream. Answer aCollection. Change the state of the receiver's write stream so that the objects contained with @aCollection are no longer accessible.

MicFwStreamWriter>>
nextWord

Answer the next word from the receiver. A word starts with a letter, followed by a sequence of letters and digits.

MicFwObject class>>
nonDescribableVariables

Answer a collection with the symbols of all variables that should not be available for extended descriptions.

MicFwPersistentObject class>>
nonDescribableVariables

Answer a collection with the symbols of all variables that should not be available for extended descriptions.

MicFwTransactedObject class>>
nonDescribableVariables

Answer a collection with the symbols of all variables that should not be available for extended descriptions.

MicFwStreamWriter class>>
on: aWriteStream

Answer a new instance of the receiver on @aWriteStream.

MicFwStreamWriter class>>
onEmptyStringWriteStream

Answer a new instance of the receiver on new write stream which works on an empty string.

MicFwProcessesBrowserMasterProcess class>>
openBrowser

MicFwProcessesBrowserMasterProcess class>>
openBrowserOn: aDomainProcessClass

MicFwNRelationship>>
orderAscending

Define the receiver to be sorted on the attributes named in @arrayOfSelectors in ascending order. The attributes are assumed to be base attributes (not relationships).

MicFwNRelationship>>
orderBy: selectorsOrClause

Define the sort order for the receiver. @selectorsOrClause can be either an array of selectors, optionally postfixed with #asc or #desc designators, or a previously created MicFwOrderClause. Answer the orderClause.

MicFwNRelationship>>
orderDescending: arrayOfSelectors

Define the receiver to be sorted on the attributes named in @arrayOfSelectors in descending order. The attributes are assumed to be base attributes (not relationships).

MicCwPanel>>
ownerView

Answer our owning view.

MicCwPanel>>
ownerView: aView

Set our owning view.

MicFwEnvironmentConnector>>
pathItems

MicFwStreamWriter>>
peek

Answer an Object which is the next accessible by the receiver's write stream, but do not make the object inaccessible. Answer nil if the receiver's write stream is atEnd.

MicFwStreamWriter>>
peekFor: anObject

Answer a Boolean which is true if the response to peek by the receiver's write stream is equivalent (=) to argument @anObject and false otherwise. If the response to peek is the same as the argument @anObject, make @anObject inaccessible, otherwise leave it accessible.

MicFwPersistentObject>>
persistenceManager

Answer the persistenceManager of the receiver.

MicMlsLCTime>>
pmString

MicMlsLCTime>>
pmString: aString

MicFwCommander class>>
portName

Answer the receiver's name, which is used as an identification to be independent of class references.

MicFwViewPort class>>
portName

Answer the receiver's name, which is used as an identification to be independent of class references.

MicFwStreamWriter>>
position

Answer an Integer representing the current position of access for the receiver's write stream.

MicFwStreamWriter>>
position: anInteger

Set the receiver's write stream position reference to argument @anInteger.

MicFwEnvironmentConnector>>
printString

MicFwViewPort>>
readAccessRejectedForAspect: anAspect vetoValue: vetoValue

Override this method to control the reaction on a rejection of a read access to the model.

MicFwPcmProcessClassesNet>>
refreshClassCollections

MicFwPcmProcessClassesNet>>
refreshProcessClasses

MicFw1Relationship>>
releaseTarget

Remove the committed target from the receiver without any transacted changes.

MicFwNRelationship>>
releaseTarget

Remove the committed target from the receiver without any transacted changes.

MicFwRelationship>>
releaseTarget

Remove the committed target from the receiver without any transacted changes.

MicFwNRelationship>>
removeAll

Obsolete. Remove all objects from the relationship maintained by the receiver. Also delete the inverse ref-

erences.

MicFwPcmProcessClassDescription>>
removeBaseConnectionDescription: aConnDesc

Remove a base connection description.

MicFwPcmProcessClassDescription>>
removeChildProcessConnectionDescription: aConnDesc

Remove a child process connection description.

MicFwPcmProcessClassDescription>>
removeDefaultBaseConnectionDescription: aConnDesc

Remove a default base connection description.

MicMlsLCMessages>>
removeKey: aKey

Accessing the internal structure.

MicMlsLocaleCategory>>
removeKey: aKey

Accessing the internal structure.

MicMlsLCMessages>>
removeKey: aKey ifAbsent: aBlock

Accessing the internal structure.

MicMlsLocaleCategory>>
removeKey: aKey ifAbsent: aBlock

Accessing the internal structure.

MicFwNRelationship>>
removeOrder

The order clause for the handled object was removed. Remove the sorted order from the target collection.

MicFwTransactionContext>>
removeVersionsFor: anAspect in: anObject

Remove all version information for <code>anAspect</code> in <code>anObject</code> in all transaction levels. Information will also be removed in child contexts. @param anObject the object containing the desired aspect @param anAspect the aspect for which to remove version information.

MicFwTransactionContext>>
removeVersionsFor: anObject

Remove all version information accumulated for <code>anObject</code>. Information will als be removed in child contexts. @param anObject the object for which to remove version information @return true when a version has been removed in one of the contexts.

MicFwStreamWriter>>
reset

Set the receiver’s write stream position reference to 0.

MicMlsLocaleContext>>
resetDescription

Reset the context description to the values stored in the class method.

MicFwCachingViewPortRequestBroker class>>
resetViewPortCache

MicFwDomainObject>>
returnContextNamed: aContextName

Return a valid context object named aContextName. If its not a master context, create a new instance

MicMlsLocaleCategory>>
returnParameterString

Return all parameters as parameter string.

MicFwPcmProcessClassesNet>>
rootModels

Return a collection with the root models of the object net.

MicFwTransactionContext class>>

running

Answer the currently running transaction context or nil if none. This is an alternative to MicFwTransaction-Manager global runningContext.

MicFwTransactionManager class>>

runningContexts

Return the running contex (or nil).

MicFwPcmProcessClassesNet>>

saveAllChanges

MicFwTypeDescription>>

scale

Answer the value of @scale.

MicFw1Relationship>>

setTo: anObject

Create the relationship connection between @anObject and #thisObject. Validate the type of @anObject and the connection, if the receiver's #validateType is <true>. Delegate to connection handler.

MicFwStreamWriter>>

setToEnd

Set the receiver’s write stream position reference to be the size of the underlying contents.

MicMlsLCDate>>

shortMonthName: aNumber

Get the short month name for <aNumber>.

MicMlsLCDate>>

shortMonthNames

Get-Accessor for shortMonthNames ^Array of String (1..12): all short month names indexed by their month number.

MicMlsLCDate>>

shortMonthNumber: aString

MicMlsLCMonetary>>

shortName

MicMlsLCMonetary>>

shortName: aString

MicMlsLCDate>>

shortWeekDayName: aNumber

Get the short week day name for <aNumber> ^String: short name of week day <aNumber>.

MicMlsLCDate>>

shortWeekDayNames

Get-Accessor for shortWeekDayNames ^Array of String (1..7): all short week day names indexed by their day number.

MicMlsLCDate>>

shortWeekDayNumber: aString

MicMlsLCDate>>

shortWeekName: aNumber

Get the short month name for <aNumber> ^String: month name in its short format.

MicCwPanel>>

showCapslockState

Show the state of the caps lock key on the view.

MicCwPanel>>

showDateToday

Show the date.

MicCwPanel>>
showNumlockState

Show the num lock key state.

MicFwStreamWriter>>
size

Answer the size of the collection over which the receiver's write stream is streaming.

MicFwTypeDescription>>
size

Answer value of instanceVariable size - generated method.

MicFwStreamWriter>>
skip: anInteger

Increment the receiver's write stream reference position by @anInteger. Fail if @anInteger is not a kind of Integer.

MicFwStreamWriter>>
skipTo: anObject

Set the receiver's write stream position reference past the next accessible occurence of the argument @anObject. Answer a Boolean which is true if an occurence of the argument @anObject is accessible, and false otherwise. If @anObject is not accessible, place the position reference past the last element of the underlying collection.

MicFwStreamWriter>>
skipToAll: aSequentialCollection

Advance the receiver's write stream position beyond the next occurrence of the elements of @aSequentialCollection or if none, to the end of stream. Answer true if the elements of @aSequentialCollection occurred, else answer false.

MicFwStreamWriter>>
space

Store the space character as next object in the receiver's write stream. Set the receiver's write stream position reference to be immediately after the space character.

MicFwViewPort>>
startInteraction: aConnection

Obsolete. Call if the process is selected via a notebook page select event. May be reimplemented to perform some actions with the process. Return true if the actions are successful, false otherwise.

MicFwPcmProcessClassesNet>>
startWithClass: aClassDesc

Generate process classes net with <aClassDesc> as source.

MicFwViewPort>>
stopInteraction: aConnection

Obsolete. Called before the the process is deselected (via a notebook page leave event). May be reimplemented to perform some actions with the process. return true if the actions are successful and the process could be deselected, false otherwise.

MicMlsLocaleCategory>>
strings

MicMlsLocaleContext>>
subcontextNamed: aLocaleName

Return a subcontext named aLocaleName.

MicMlsLocaleContext>>
superContext

Returns the super context of self.

MicMlsLCMonetary>>
symbolName

MicMlsLCMonetary>>
symbolName: aString

MicFwStreamWriter>>
tab
Store the tab character as the next object in the receiver's write stream. Set the receiver's write stream position reference to be immediately after the tab character.

MicMlsLocaleCategory>>
territory

MicMlsLocaleCategory>>
territory: aTerritory

MicMlsLCKeys>>
testResultWithKey: aKey
Answer a test result using a key.

MicFwEnvironmentConnector>>
title

MicFwJavaConnector>>
title

MicFwNRelationship>>
transact
Answer true if the relationship access handler handles transacted accesses, false otherwise.

MicFwTr1Relationship>>
transact
Answer true if the relationship access handler handles transacted accesses, false otherwise.

MicFwTransactedObject>>
transact
Determine whether the receiver should be transaction handled.

MicFwNRelationship>>
transact: aBoolean
Set the transaction property for the receiver. If @aBoolean = true -> handle accesses transacted.

MicFwTr1Relationship>>
transact: aBoolean
Set the transaction property for the receiver. If @aBoolean = true -> handle accesses transacted.

MicFwTransactedObject>>
transact: aBoolean
Set whether the receiver should be transaction handled.

MicFwTransactionContext>>
transactionLevel
Return the active (highest) transaction level of receiver (as Integer).

MicFwStreamWriter>>
truncate
Set the size of the receiver's write stream to its current position.

MicFwTypeDescription>>
type
Answer value of instanceVariable type - generated method.

MicMlsLocaleCategory>>
unicodeStrings

MicFwStreamWriter>>
upTo: anObject
Answer a SequenceableCollection of objects starting with the next element accessed by the receiver's write stream, and up to, not inclusive of, the next element that is equivalent to argument @anObject.

If argument @anObject is not within the remaining accessible objects of the receiver's write stream, then answer a SequenceableCollection containing all the remaining accessible objects. Adjust the receiver's write stream position reference to the next element after @anObject.

If no elements equivalent to @anObject are accessible, adjust the receiver's write streamposition reference to after the last element of the underlying collection.

MicFwStreamWriter>>
upToEnd

Answer the elements between the current position and the end of the stream, inclusive.

MicFwDomainObject>>
validateWrite: #var using: aBlock

Causes any attempted assignment of an object to the variable var to be allowed only if the object to be assigned meets the criteria specified in the block aBlock.

MicMlsMonetaryObject>>
value

MicMlsMonetaryObject>>
value: aValue

MicMlsLCDate>>
weekDayName: aNumber

Get the week day name for <aNumber> ^String: day name.

MicMlsLCDate>>
weekDayNames

Get-Accessor for weekDayNames ^Array of String (1..7): all week day names indexed by their day number.

MicMlsLCDate>>
weekDayNumber: aString

MicMlsLCDate>>
weekStartDay

MicMlsLCDate>>
weekStartDay: anInteger

1..7.

MicCwPanel>>
windUpClock

Update time in the interval of 1 second.

MicFwPcmProcessClassesNet class>>
with: aSource with: aDestination

Generate environment and process classes net with <aClass> as source.

MicFwPcmProcessClassesNet class>>
with: aSource with: aDestination with: aClass

Generate environment and process classes net with <aClass> as source.

MicFwViewPort>>
writeAccessRejectedForAspect: anAspect vetoValue: vetoValue

Override this method to control the reaction on a rejection of a write access into the model.

MicFwStreamWriter>>
writeStream

Answer the receiver's write stream.

MicFwStreamWriter>>
writeStream: aWriteStream

Set the receiver's write stream to @aWriteStream.

Appendix B

Glossary

This glossary defines terms that are presented throughout this manual.

->1 relationship: In ->1 relationships, a source instance variable references 0..1 target objects (nil or 1 object). No target instance variable refers to the source object.

An example of a ->1 relationship is the relationship between Person (source) and PersonName (target). Person instance variable name references 1 PersonName object (a person has only 1 name). No PersonName instance variable references the Person object (a PersonName object never needs to know which Person object is referencing it).

->N relationship: In ->N relationships, a source instance variable references min...max target objects (where min...max is specified by the cardinality).

An example of a ->N relationship would be the relationship between Person (source) and PersonName (target), with the assumption that a person can have more than 1 name. Person instance variable name references min...max PersonName objects, where min = 1 and max is unspecified. No PersonName instance variable references the Person object.

1<->1 relationship: In 1<->1 relationships, a source instance variable references 0..1 (signified by the "1" on the RIGHT of "1<->1") target object. The target instance variable references the same 1 (signified by the "1" on the LEFT of "1<->1") source object or nil. The relationship is not primitive, because the target object should have a reference to the source object.

An example of a 1<->1 relationship would be the relationship between Customer (source) and Portfolio (target). Customer instance variable portfolio references 1 Portfolio object. Portfolio instance variable customer references the 1 Customer object. The relationship is not primitive, because a Portfolio object should know which object is its Customer.

1<->N relationship: In 1<->N relationships, a source instance variable references N (where N is any number with the range max...min specified by the cardinality of the relationship) target objects. There is an instance variable in all of the referenced target objects that reference the 1 source object.

An example of a 1<->N relationship would be the relationship between Employee (source) and Customer (target). Employee instance variable ownedCustomers would reference N (cardinality min <= N <= cardinality max) Customer objects. The instance variable ownerEmployee in each referenced Customer object would reference the 1 Employee object.

Abort a context: A context is aborted when all version objects within the context are dereferenced (ie, none of the changes that were transacted while the context was active will actually be implemented) and the context ceases to exist. See: abortcontext.

Abstract Control:: An Abstract Control simplifies external access to the Framework, as such access can only take place via real Control. There are only a small number of genuinely different Abstract Controls (see command example in the MVC chapter). The actual access attempts are handled with real Control via an appropriate Adapter.

Focus change, issuing commands and transfer of data - including a range of state information like validation or authorization - will be done in this abstract layer. The real Controls are completely decoupled from all Domain Process actions and Domain Model data; all technical details of external interfaces (GUI, DDE, ...) are completely hidden in the Abstract Control implementation.

Abstract Event: Abstract Window Events are the objects that really perform the communication between the view system and the model world, whereas Abstract Windows are merely containers for Abstract Events which additionally may provide some services for them. Abstract events can be divided into two functional types: Abstract events which propagate changes in the model world to the view and Abstract Events which propagate changes or requests from the user interface to the model world.

Abstract Value: An Abstract Value is a container object that is used by Viewports to keep and propagate information about a Viewport Aspect to and from the view. It does not only contain a value for the content of a control, but also different kinds of state information like whether or not the control should be enabled, what the (background) color is, which context help text is displayed, and so on.

Abstract View: When a real View is created, an Abstract View begins to exist in its shadow. The lifetime of the Abstract View is exactly determined by the lifetime of the real View. The Abstract View's purpose is to manage the Abstract Controls corresponding one-to-one to the real Controls on the view. Moreover, the Abstract View performs coordination between abstract and platform layer when opening, activating and closing the view. It communicates with the real Platform View using a Platform Adapter.

Accessor Generator: The accessor generator is the object that creates the OBF accessors for a variable.

Active Context: Only 1 context can be active at anytime. While a context is active, any changes to any trans-

acted variables will be recorded in object versions. This variable will be locked by the context, which means that no changes may be made to the variable while a sibling context or parent context is active as long as this context exists.

Adapter: A Platform Adapter system is introduced which does the translation of protocols and overcomes the architectural differences between the host Smalltalk system architecture and the Application Framework architecture. This makes the core part of the framework itself portable.

Archiver: See: Code Archiver.

Authorization: The access to individual objects controlled on the model level. Authorization works both for accessing attributes of Domain Objects and for executing Aspects of business processes.

Broker: To allow subsystems or specific requests to them to be exchanged with an own implementation, Application Framework uses Broker classes that offer a thin public interface with the internal knowledge how to delegate the call to the subsystem. A common Broker concept enables the developers to modify request algorithms or subsystem behavior quite easy.

Cardinality: The cardinality of a relationship determines the required min and max number of target objects referenced by the source variable. In 2-way relationships, there are 2 cardinalities. The second cardinality determines the requried min and max number of source objects referenced by the target variable.

CB: Connections Browser.

Child context: A child context can change any variable locked by its parent. A variable, having been changed by the child context, is now locked by the child context. The parent context cannot change a variable locked by the child. See: parent Context.

Code Generator: See Accessor Generator.

Commit a context: Committing a context has the same effect as committing all transaction levels (TrLevels) in the context.

Committed target: In a VersionObject: A getter message to a source object will return the committed target object referenced by the variable if no context is active OR if [the active context read mode is isolate AND the source object's variable is not locked by the active context AND the active context is not a child context of the context with the variable lock]. See: transacted Target.

Concurrent contexts: 2 contexts are concurrent if there is no parent-child relationship between them. Such contexts can also be referred to as "sibling" contexts. A sibling context may not change a variable locked by another sibling context.

Concurrent transactions: See Concurrent Contexts.

Connector: A Connector is used to connect objects (Domain Objects and Domain Processes) to an external view. The Connector decouples view and model objects and provides access to connected Domain Objects and Domain Processes on a low level. In this respect, it forms a bridge between the objects, decoupling also Domain Objects from Domain Processes and thus making them more combinable and exchangeable.

Context: See Transaction Context.

DDL: A language enabling the structure and instances of a database to be defined in a human- and machine-readable form.

Default Base Connection: The Default Base Connection will be used to hold the Domain Object if no explicit Base Connection is defined for this process.

Delegation Model: An concept used to decouple subsystems from each other. Application programmers implement subclasses of Viewport to isolate the model from the interaction subsystem.

Domain Model: A Domain Model is a design phrase for real world concepts like Customer, Policy or Address. Its instances are called Domain Objects.

Domain Object: Domain objects describe that part of the MVC architecture which corresponds to business terms. The behavior and structure are primarily defined in this respect. The appropriate tool (object net browser) from the Object Behavior Framework is used for this purpose, and mapping to the database is described with STOPF from Persistence Framework . The Domain Object also has open interfaces for connecting authorization and validation.

Domain Process: A Domain Process is a design phrase for processing and workflow-oriented tasks and control flow. Its instances are also called Domain Processes. Due to their controlling-oriented nature, they take also responsibility for managing a Transaction Context if appropriate.

Domain Processes Browser: A tool, supplied with the Application Framework to browse the exiting Domain Processes

DPB: Domain Processes Browser.

Extended Description (MicFwExtendedDescription): The object that completely describes the typing of all variables in a class. The class method createExtendedDescription creates the MicFwExtendedDescription object.

Framework Logger: The tracing facility of the Framework's. Events and messages will be routed through this logger.

Framework: A Framework is a software architecture for certain tasks, which components can be easily reused by application developers. It provides the system with a basic structure in being a collection of cooperating and conceptual concise classes and methods, which are designed to support a task-oriented work progress in application development.

Inactive Context: A non-active context. See: Active Context.

InstanceVariableDescription (MicFwInstanceVariableDescription): The object that completely describes the typing of a single instance variable in a class. The class method createExtendedDescription creates the MicF-wInstanceVariableDescription object (which is referenced within the MicFwExtendedDescription object).

Isolated Context: A variable getter sent to a source object that is locked by another context while an isolated context is active will return not the committed target of the transaction object in the other context, but rather the uncommitted target. See: uncommitedread context.

M<->N relationship: In M<->N relationships, a source instance variable references M (where M is any number with the range max...min specified by the cardinality LEFT of the relationship) target objects. There is an instance variable in all of the referenced target objects that reference the N (where N is any number with the range max...min specified by the cardinality RIGHT of the relationship) objects of the source object class, including the current source object.

An example of an M<->N relationship would be the relationship between Person (source) and Address (target). Person instance variable addresses would reference M (cardinality min <= M <= cardinality max) Address objects. Address instance variable persons would reference N (cardinality min <= N <= cardinality max) Person objects, including the current Person object.

Mapper: To maintain registration of loosely coupled elements within the Framework, Mappers are used to set, get and remove associations between unique, constant names and their corresponding classes, which themselves may change or may be reimplemented in your project. Mappers are implemented as singletons and can be reached through an easy to use interface as they extend Object with one method per Mapper.

MicFwTransactionContext:

Model View Connector: Is the object that holds the child-process- and Base-Connections for one process.

Model View Controller: The MVC (concept of a Model View Controller) defines an architecture intended to yield a strict decoupling of Domain Model Aspects, flow control and external views - mostly displayed graphically for user interaction.

Nested transaction levels: The transaction levels in a context are nested if more than 1 level exists.

OBF (Object Behavior Framework): OBF is a library of classes that when added to your Smalltalk development environment provides a Framework for defining the requirements and restrictions for the behavior of objects.

Object Net Browser: A visual OBF tool that dispayes both the object net of a class and information about the instance variables of the class (key, validated, transacted, persistent, type, relationship). The visual OBF tools Type Editor and Relationship Editor are opened from the Object Net Browser.

Object Net: An Object Net consists of objects and its relationships between them.

ObjectVersion (version object): An ObjectVersion is created if a transaction context is active and a target object is assigned to a transacted variable of a source object. The ObjectVersion is assigned to the highest TrLevel in the active context. If an ObjectVersion already exists in this TrLevel, it is replaced. The ObjectVersion references the committed target object (committed) and the uncommitted target object (uncommitted) referenced by the variable.

ONB: Object Net Browser.

Packaging: The process of creating a runtime executable.

Parent context: A parent context is a context object that created its child context when it responded to the

newTransactionContext message. The parent context can have any number of child contexts. A parent context can also be a child context. If parent context B has a parent context A and B also has a child context C, then A and C also have a parent child relationship. See: Child Context.

POM: Persistent Object Manager.

Primitive relationship: A 1-way relationship. See: 1-way relationship.

Relationship Editor: A visual OBF tool for establishing relationships between 2 classes. See: object Net Browser.

Relationship: In Smalltalk, relationships between objects are simply represented as object pointers. No distinction is made between complex and simple (scalar) data types since all are full-scale objects and there is conceptually no difference between a relationship and an "embedded" value.

Beyond this, the Mynd Object Behavior Framework provides an elaborated relationship concept maintaining referential integrity between objects. Those relationships may even be mapped to (relational) databases with the help of the Mynd Persistence Framework

RelationshipDescription (MicFw~): Object which contains complete information about a relationship.

Running context: A context that has a TrLevel1 (it may have higher levels also).

Sibling contexts: See Concurrent Contexts.

STOPF / ODBC: Smalltalk Object Persistence Framework

Transacted variable: A variable that has been marked as transacted in the Object Net Browser. NOTE:Changes to a variable will not be transacted even if a transaction context is active if the variable is not marked as transacted.

Transaction Browser: A Visual OBF tool for displaying and manipulating (aborting / committing) transactions.

Transaction Context: A logical unit that can contain transaction levels and that can be related to other contexts as a parent, sibling, or child.

Transaction Level: A subunit of a transaction context. A transaction level has a single object version for each variable of each object that was assigned a new target object while the the transaction level was the highest level in its context and its context was active.

Transaction Manager: The transaction manager is a single instance (singleton) of MicFwTransactionManager and manages all running contexts.

Transaction: A defined state of an object that runs under transactional control will be stored. If the interaction on this object fails by some reason, this object can be restored to this state, if the interaction succeeds, the persistent state of this object will be modified to this new state and the old state will be dropped.

TrLevel1, 2, ...: Transaction level 1, 2, ...

Type Converter: A type converter is assigned to a typed variable (including variables in a relationship). When an object is assigned to the variable that is not of the type specified for the variable, the type converter will be used to attempt to create the correct type of object containing the information in the original object and assign this correct type of object to the variable.

Type Editor: A visual tool for setting a variable type as a simple type (Integer, Date, etc.). The type converter and size / scale for the variable type can also be selected. The Type Editor is opened from the Object Net Browser.

TypeDescription (MicFw~): Object which contains complete information about the typing of an instance variable.

Typing: The term typing defines the assignment of types (normally basic classes like Integer or String) to instance variables. Since Smalltalk is an untyped language, the Mynd Object Behavior and Persistence Frameworks introduce typing in order to support storing of objects into (typed) relational databases.

UML (Unified Markup Language): Graphical notation for OO analysis and design.

uncommitedRead context: If an ObjectVersion exists for aSourceObject>>aSOVariable (ie, aSOVariable is transacted and anUncommittedTargetObject was assigned to aSOVariable while aContext1 was active): If aContext2 is an active uncommitedRead context and getter message sOVariable is sent to aSourceObject, the object returned will be anUncommittedTargetObject. See: Isolated context.

Note: The message is spelled "uncommitedRead".

Uncommitted target: In a VersionObject: A getter message to a source object will return the uncommitted target

object referenced by the variable if [the active context read mode is uncommitedReadisolate AND the active context does not have a lock on the source object variable AND the active context is not a child context of the context that has the lock] OR if the source object's variable is locked by the active context. See: Committed Target.

Validation of ExtendedDescription: Validation of an ExtendedDescription involves checking the Object Net for errors.

Validation: Like the authorization service, validation ties in at the model layer and is used for checking value-based access to and from attributes based on certain rules.

VariableDescription (MicFw~): Object which contains complete information about an instance variable.

Viewport: In order to separate view-related state handling from Domain Objects and Domain Processes, Application Framework implements a Viewport as a Delegation Model concept. Functionality for transportation and the filtering of data and states concerning an object net from a certain object's or view's direction is delegated from the Domain Models to them, leading to a more lightweight and view independent kind of Domain Object.

Viewports "learn" the dependencies between their Aspects and the corresponding Model Aspects while running the application. This read trace frees the programmer from dealing with changed-events and allows Application Framework to update both sides automatically and in a generic way.

A Dispatcher is called Viewport if Domain Objects are concerned.

Bibliography

The following documents were referenced throught this document.

[APPFW] Mynd Application Framework Manual

[CANN] Cannan, Otten. The SQL Standard Handbook, McGraw Hill, 1993

[ODBC20] Microsoft ODBC 2.0 Programmers Reference and SDK Guide. Microsoft Press, 1994

[OBFW] Mynd Object Behavior Framework Manual

[CODD70] E.F. Codd, A Relational Model of Data for Large Shared Data Banks, CACM 13, No. 6, June 1970

[CODD90] E.F. Codd, The Relational Model for Database Management, Version 2, Addison Wesley, 1990

[DATE] C.J. Date, An Introduction To Database Systems, 5th Edition, Addison Wesley, 1990

[SSQL] Static SQL for Smalltalk and DB2 Databases Programming Guide, Version 2.0, Neva Object Technology, Inc.

[Watcom SQL] Sybase SQL Anywhere, User's Guid Vol. I/ II, A System 11 Server Product

List Of Tables

Table 1: Obsolete classes. 22
Table 2: Supported relational databases for operating system environments 27
Table 3: Configurable items for relational persistent object managers 35
Table 4: Description of relations 47
Table 5: Description of objects 47
Table 6: Example for subtyping 77
Table 7: Abbreviations for the chapter User-defined relationships 82
Table 8: Non-column-mapping variables for user-defined-relationship 84
Table 9: Behavior of deletions (before final commit of the context) 98
Table 10: Behavior of deletions (after final commit of the context) 99
Table 11: Operators for query conditions 138
Table 12: Operand types for query conditions 139
Table 13: Class hierarchy for supported SQL functions. 145
Table 14: List of error classes for queries 150
Table 15: ESQL methods generated for SQL statements. 166
Table 16: Configuration maps for PFW cross development support 178
Table 17: List of validations currently supported 195
Table 18: List of error messages from the POM generator 212
Table 19: List of warning messages from the POM generator 212

List Of Figures

Figure 3: Persistent class derived from MicFwPersistentObject 32
Figure 4: Persistence Manager Hierarchy 35
Figure 5: Static and dynamic POM components 39
Figure 6: Relational Tables with Primary and Foreign key 42
Figure 7: Identifying Relationship 43
Figure 8: Foreign Key Relationship 44
Figure 9: N:M relationship using an associative relation 44
Figure 10: Object Model 46
Figure 11: The Customer Base Management class model 55
Figure 12: Relational model Customer Base Management 56
Figure 13: The main STOPF dialog window 58
Figure 14: The 1:1 Relationship dialog window 60
Figure 15: The N:M Relationship dialog window 61
Figure 16: The Aggregation Relationship dialog window 61
Figure 17: UDR Static SQL Clause left specification 62
Figure 18: UDR Static SQL Clause right specification 63
Figure 19: UDR Specification Dialog 63
Figure 20: Attribute settings for MicFwXmpProduct>>timestamp 65
Figure 21: Attribute settings for MicFwXmpProductgroup>>timestamp 65
Figure 22: POM Configuration 66
Figure 23: QueryRegistration #getServerTimestamp 66
Figure 24: Persistent-Tutorial MicPsTutorialPersonListEditor 69
Figure 25: OQL-Browser 69
Figure 26: OQL-Browser after execute a sample query 70
Figure 27: Canonically Mapped Class Hierarchy 75
Figure 28: Denormalized Mapping of Class Hierarchy 75
Figure 29: Single class spanning several tables 78
Figure 30: Foreign Key Based Relationship 79
Figure 31: Many-to-Many Relationship 80
Figure 32: Aggregated Relationship 81
Figure 33: CONTRACT (versioned) and CUSTOMER 82
Figure 34: Mappings for Account-Currency Relationship 86
Figure 35: Example of a class hierarchy retrieval 105
Figure 36: Persistence Frameworks application layers 117
Figure 40: Object Query Components 136
Figure 41: Pessimistic vs. Optimistic Concurrency 153
Figure 42: Duplicate primary key conflict 154
Figure 43: Deleted object causing an update conflict 154
Figure 44: Concurrent change causing an update conflict 154
Figure 45: Concurrent change causing a delete conflict 155
Figure 46: ESQL Persistence Components 166
Figure 47: Precompilation Dialog 170
Figure 48: Parameter Select Statement Editor 171
Figure 49: Main view of the STOPF tool 184
Figure 50: Browse Persistence Manager dialog 185
Figure 51: The Manager menu 185
Figure 52: Persistence Manager Options 188
Figure 53: Statement and query registration 190
Figure 54: Query editor 191
Figure 55: Order Clause Editor 191
Figure 56: Order clause registration 192
Figure 57: The Class menu 196
Figure 58: The Variable menu 197
Figure 59: The 1:1 Relationship dialog window 198
Figure 60: The N:1 Relationship dialog window 198
Figure 61: The N:M Relationship dialog window 199
Figure 62: Aggregation editor 200

Figure 63: Relationship editor for user-defined relationships ... 201
Figure 64: Editor for user-defined-relationship mappings ... 201
Figure 65: Dynamic specification editor ... 202
Figure 66: Static SQL clause specification editor ... 202
Figure 67: Column mapping attributes editor ... 203
Figure 68: The Schema menu ... 204
Figure 69: Foreign Key Browser ... 205
Figure 70: Foreign key editor ... 206
Figure 71: Mapped Class Generator ... 208
Figure 72: POM generator dialog window ... 209
Figure 73: Suppress table ... 210
Figure 74: Denormalize table ... 210
Figure 75: Menu Generator ... 211
Figure 76: Menu Error Log ... 211

Index

A

Abstract Control 243
Abstract Event 243
Abstract Value 243
Abstract View 243
AbtDatabaseSupport 27
access set 179
Adapter 243, 244
aggregation 34, 200
allDo 109, 110
answerSetSize 106, 107, 190
API . 35, 38, 47, 50, 93, 94, 102, 106, 110, 113, 117, 118, 124, 126, 127, 144, 161, 187, 189, 190, 196, ... 209, 213
applheapsz 169
architecture 245
asESQLString 169
authorization 244, 247

B

Base Connection 244
blockSize 110, 111
Broker 244

C

Cached relationships 86
cached statements 189
cachedObjects 86, 87
Changing Tables 204
child class 79
Class mapping 34, 204
Class Storage 186
combinable 244
Commit Handler ... 35, 158, 160, 187, 203, 209
compileMappings 196
Concurrent changes 154, 159, 162
Configuration 34, 65, 88, 95, 157, 160, 184, 187
conflicts 49, 103, 153, 158
Connector 244
control flow 244
createExtendedDescription 196

D

Database connection 35, 57, 95, 155, 213
database schema 33, 57, 62, 66, 81, 107, 184, 194, ... 195, 202, 204, 205, 206, 207, 209, 210
Database Transaction 155
database trigger 88, 157
DB2 22, 25, 27, 66, 94, 100, 157, 158, 177, 179, 187
Default Base Connection 244
Define Subtype Column 59, 196
Define Type Value 59, 196
defineVariable 142, 143, 146, 148, 149
Delegation Model 244
delete 23, 24, 35, 37, 43, 69, 80, 98, 99, 100, 107, 109, 124, 155, 156, 159, 187, 190, 195, 197, 200, 205, ... 206
deleteAllInstacesOf 107
Deleted objects 154, 158
deleteInstances
where 107
deleteObject 37, 101, 107
deleteWithReferences 107
Domain Model 244
Domain Object 244
Domain Process 244
Domain Processes Browser 245
Dump to file 186

E

Embedded SQL 126
Errors 68, 81, 115, 150, 159, 193, 209
Exceptions 104, 110, 111, 159, 162, 199
executeImmediate 169, 171
Export to file 21, 186
extent identifier 36, 104

F

Fileout Storage 186
foreign keys 197, 198, 199, 205
Framework 245
Framework Logger 245

G

Generator 22, 23, 126, 207, 208, 209, 211
generator . 21, 48, 64, 124, 126, 127, 207, 208, 209, ... 211, 212
getLoaded 63, 84, 107
globalPOM 94

I

Import tables 204
initializer class 22, 94, 186
installing 27
Instance comparison 156
Integrity Violation 44, 115, 155
interaction 244
Internal Table Connection 59, 77, 78, 197

isConnected... 67, 95
Isolation Level... 153
Iterators.................................... 109, 110, 118

L

Load Feature 28
loadedSize 107
Locking 64, 111, 153
Logger.. 245

M

Mapper.. 245
Mapping 33, 41, 47, 57, 59, 69, 81, 86, 88, 107, 114, ...156, 160, 184, 193, 199, 203, 207
markToBeCompiled..................... 85, 105, 106
MicFwPersistenceConflict 159
MicFwPersistenceError 159
MicFwPersistenceManager. 25, 34, 35, 39, 94, 95, .98, 101, 103, 104, 107, 158, 184, 185, 190
MicFwPersistenceManagerInitializer ... 22, 186
MicFwPersistenceManagerOdbc 25, 57, 67, 94, 96, ...196
MicFwPersistenceManagerRdb 35, 94, 98, 101, 105, ...159, 161, 190, 196, 213
MicFwPersistenceManagerSSQL 25
MicFwPersistentObject 32, 35, 50, 58, 103, 107, 108, ...114, 115, 116, 159, 161, 162, 207, 209
MicFwPsAspectCompareConflict....... 159, 161
MicFwPsCompareConflict.......... 115, 159, 161
MicFwPsDeletedConflict 159, 161
MicFwPsExtentIdentifier 21, 36, 109, 113, 117, 135
MicFwPsExtentIdentifierSQL . 21, 36, 106, 117
MicFwPsIterator....................................... 110
MicFwPsProjectionDescriptor 21, 102, 117, 140
MicFwPsUpdateConflict 115, 159
MicFwPsUpdateError 68, 115, 159
MicFwRdbCmtHdlrOptimistic 65, 160, 161, 187
MicFwRdbCommitHandler 160, 187
MicFwRdbDirectStatement 21, 66, 102, 117, 190
MicFwRdbIntegrityError..................... 115, 159
MicFwRdbOrderClause 21, 117
MicFwRdbRTMappingInfos 21, 124
MicFwRdbTableIdentifier........................... 204
MicFwSQLFunction 144, 145
MicFwStatementSettingsBrowser 189
MicOvmObjectVersion............................... 108
model 244
Model View Connector 245
Monitor target 87
MVC.. 244, 245
MVS Smalltalk 12

N

newValueFor 108
nextBeginQuery . 142, 143, 145, 146, 148, 149
nextEndQuery 142, 143, 146, 147, 148, 149
nextPutAnd........................ 143, 146, 147, 149
nextPutComparison ... 142, 143, 146, 147, 149
nextPutConstant......... 142, 143, 146, 147, 149
nextPutFunction 145, 146
nextPutMessages 142, 143, 146, 147, 149
nextPutVariable.......... 142, 143, 146, 147, 149
Notifications..................................... 115, 159

O

Object Behavior Framework 244, 246
Object Dump Storage 186
object net.. 245
Object Net Browser................................... 244
Object-Query-Language 12, 23
ODBC 27, 37, 38, 57, 94, 95, 96, 110, 111, 114, 115, ...117, 184, 185, 187, 188, 189
OdbcObject 95
ODMG93 .... 31, 36
oldValueFor... 108
Optimistic column............................. 156, 194
optimistic concurrency .. 64, 88, 104, 111, 115, 158, ...161, 187, 204
Optimistic strategy 64, 88, 156, 160, 203
Optimistic timestamp column.................... 156
Oracle......... 22, 27, 94, 97, 116, 117, 185, 187
order clause 189, 190, 191
orderBy.................................... 105, 110, 113

P

packaging... 245
parameter marker 84, 106, 203
parent class 79
persistence context...... 67, 103, 107, 158, 187
Persistence Framework 11, 53, 57, 93, 98, 102, 106, 115, 156, 159, 193, 195, 202, 207, 244, 246
Persistence Manager 35, 57, 67, 81, 88, 94, 98, 110, ... 111, 115, 156, 160, 184, 186, 200, 201
persistenceDelete 108
persistenceInitialize.................................. 107
persistenceInsert............. 63, 64, 84, 107, 108
persistenceLoaded.................................... 108
persistenceUpdate 63, 64, 84, 108
Persistent Object 25, 33, 34, 48, 79, 88, 98, 104, 107, ... 110, 154, 155, 158, 159, 162, 183
Pessimistic concurrency 153, 187
Platform Adapter 243
polymorphic load 76
POM 21, 22, 23, 25, 33, 34, 38, 39, 48, 49, 50, 57, 58, 61, 64, 65, 66, 67, 69, 76, 81, 83, 86, 88, 94, 95, 96, 97, 99, 100, 103, 107, 108, 110, 114, 115, 116, 117, 118, 123, 124, 125, 126, 127, 156, 158, 160, 177, 178, 179, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 195, 196, 197, 199, 202, 203, 204, 205, 206, 207, 208, 209, 211, 212, 213, 214

MYND

portable 244
Prefetch 66, 158
pregenerated 189, 190
primary key .. 42, 47, 75, 86, 88, 100, 103, 107, 115, ... 153, 156, 194, 195, 203
processing 244

R

read trace 247
real Control 243
real View 243
real world concepts 244
reentrancy 21, 117
referential integrity 246
refresh 35, 81, 88, 155, 159, 161
relational databases 246
relationship 245, 246
release. 21, 23, 24, 39, 50, 109, 111, 118, 119, 124, ... 126
Remove conflicting mappings 204
Remove Table 204
removeLoaded 107
RT-POM 21, 117, 123, 124, 125
runtime executable 245

S

Schema report 204
setSingleObjectMode 111
size 23, 38, 106, 107, 110, 111, 112, 114, 123, 126
Sorting for relationships 113
Statement Registration 24, 66, 189
Statement Registration Tool 189
statements 189, 192
Static SQL 62, 89, 110, 201
STOPF. 12, 47, 57, 61, 64, 65, 83, 86, 94, 156, 158, 183, 189, 193, 195, 200, 204, 206, 211, 244
Store in class 21, 186
supportsExtendedFetch 110
Sybase SQLAnywhere 27, 57

T

tool-POM 48, 123, 124, 125, 127
transaction 246
Transaction Context 244
transaction context 32, 35, 36, 98, 109, 115, 155, 159
Type Discriminator 77
type discriminator 75, 77, 196
type value 59, 77, 194
typing 246

U

UDR 64, 84
Unified subtable load 23, 75, 76
untilEndDo 36, 109, 110, 118, 119
update 247
User-defined relationship 200
User-Defined-Relationship 62, 64

V

validation 244, 247
Value strategies 64, 88, 156, 203
Viewport 243, 244, 247

W

workflow 244