

VisualAge Smalltalk



ObjectExtender User's Guide and Reference

Version 5.5

Note

Before using this document, read the general information under "Notices" on page v.

August 2000

This edition applies to Version 5.5 of the VisualAge Smalltalk products, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. The term "VisualAge," as used in this publication, refers to the VisualAge Smalltalk product set.

Portions of this book describe materials developed by Object Technology International Inc. of Ottawa, Ontario, Canada. Object Technology International Inc. is a subsidiary of the IBM® Corporation.

If you have comments about the product or this document, address them to: IBM Corporation, Attn: IBM Smalltalk Group, 621-107 Hutton Street, Raleigh, NC 27606-1490. You can fax comments to (919) 828-9633.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1998, 2000. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	v
Trademarks.	v

About this book.	vii
What this book includes	vii
Who this book is for	vii
About this feature	vii
Conventions used in this book	vii
Tell us what you think	viii

Chapter 1. Introduction	1
Minimal intrusion to object and database design	2
High performance	2
Advanced transaction support	4
Advanced query support	5
Relationship support.	6
Seamless support for various database paradigms	6

Chapter 2. Concepts	9
Key elements managed by the framework	9
Key tasks you develop	9
Organizing your application.	10
The main components of the framework.	11
Model framework	12
Relationship framework	12
Transaction framework	13
Mapping framework	14
Data store framework	16

Chapter 3. Quick tour	19
Choosing an approach to persistence	19
Using code generation services	19
Defining the model with the Model Browser	20
Importing a model from UML Designer	22
Defining the schema	22
Defining a data store map	24
Tuning the code as needed	25
Completing your application	25
Exporting your model to UML Designer.	25

Chapter 4. Tools.	27
The Model Browser.	27
Browser use	27
Browser description	28
Browser menu-bar choices	29
The Schema Browser	32
Browser use	32
Browser description	33
Browser menu-bar choices	33
The Map Browser	38
Browser use	38
Browser description	38
Browser menu-bar choices	39
The SQL Query Tool	41

The Status Tool	42
---------------------------	----

Chapter 5. Tasks	43
Tasks and samples overview.	43
Your first ObjectExtender application	43
Creating a model	43
Creating a class in the model	45
Creating the code for the model	45
Creating persistence support.	46
The department home collection class	47
Creating a view	49
Listing departments	49
Editing departments	49
Creating departments	51
Deleting departments	52
Creating a new top-level transaction each time one is committed	53
Transactions in more depth	55
Nested transactions.	55
Switching between transactions.	57
Two top-level transactions	58
Visual programming for more than one transaction	58
Using the TransactedVariable part	59
Viewing multiple transactions	61
Transacted variables in editable container parts	62
Model to model relationships	63
One-to-many relationship.	64
Maintaining staff	65
Maintaining the department to staff relationship	68
Creating relationships	69
Creating one-to-one (1-1) relationships	69
Creating one-to-many (1-M) relationships	72
Creating many-to-many (M-M) relationships	74
Mapping business objects to tables	78
Creating a single table map with no inheritance	78
Creating a secondary table map	78
Creating single table inheritance maps	79
Creating root/leaf inheritance maps	80
Using a composer for mapping an attribute to multiple database fields	83
Using converters	84
Performance tuning.	85
Changing the locking type on a table.	85
Setting preload paths	86
Creating Lite collections	86

Chapter 6. Reference	89
ObjectExtender runtime architecture	89
Programming model overview	89
The business object layer	90
The persistence layer	102
Metadata storage model.	108
Writing your own services	110
Defining the persistence support code	110
User code sections.	110

Managing business objects	111
Using the data store	112
Creating, retrieving, deleting instances	112
Accessing relationships of a business object	113
Coding transactions manually and visually	113
Analyzing performance	115

Appendix. Restrictions	117
Glossary	123
Index	125

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, USA 10594.

IBM may change this publication, the product described herein, or both.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

- AIX[®]
- DB2[®]
- CICS[®]
- IBM
- IMS[™]
- OS/2[®]
- VisualAge[®]

The following terms are trademarks of other companies:

Acrobat Reader	Adobe Inc.
Microsoft [®]	Microsoft Corporation
Netscape	Netscape Inc.
Windows [®]	Microsoft Corporation
UNIX [®]	X/Open Company Limited

Windows is a trademark of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

About this book

This book contains introductory, sample, and reference information for the VisualAge Smalltalk ObjectExtender feature.

What this book includes

The opening chapters contain high-level conceptual material followed by a brief tour of the tasks you perform when using the feature. Samples are provided next to guide you from simpler to more complex tasks. Finally, reference information provides some of the lower-level details of the various framework components.

In addition to this book, be sure to check for feature updates on the web at <http://www.software.ibm.com/ad/smalltalk>. Code for the samples will be available for download from the site. Use the keywords, ObjectExtender, or samples from the home page to search for any updates.

Who this book is for

This book is written for VisualAge application developers, database access programmers, and object modelers, who need to develop a persistence layer for large-scale enterprise applications. It is assumed that you are familiar with developing applications using the VisualAge development environment. It is helpful to be familiar with the concepts of persistence and transactions, however, sufficient conceptual information is provided for those who may be addressing these concepts for the first time.

About this feature

The *VisualAge Smalltalk ObjectExtender Guide* describes the ObjectExtender feature: a powerful and extensive persistence framework. ObjectExtender enables your object models to persist in relational data stores as well as provides linkages to legacy data on CICS and IMS systems. Creating the persistence layer is accomplished using the ObjectExtender tool set. The tool set helps you describe the business objects in your model that will persist in a data store. The tools generate the supporting code that services your persistent business objects.

Conventions used in this book

This book uses several conventions that you might not have seen in other product manuals.

Tips and environment-specific information are flagged with icons:



Shortcut techniques and other tips



VisualAge for OS/2



VisualAge for Windows



VisualAge for UNIX platforms

These highlighting conventions are used in the text:

Highlight style	Used for	Example
Boldface	New terms the first time they are used	VisualAge uses construction from parts to develop software by assembling and connecting reusable components called parts .
	Items you can select, such as push buttons and menu choices	Select Add Part from the Options pull-down. Type the part's class and select OK .
<i>Italics</i>	Special emphasis	Do <i>not</i> save the image.
	Titles of publications	Refer to the <i>VisualAge Smalltalk User's Guide</i> .
	Text that the product displays	The status area displays <i>Category: Data Entry</i> .
	VisualAge programming objects, such as <i>attributes, actions, events, composite parts, and script names</i>	Connect the window's <i>aboutToOpenWidget</i> event to the <i>initializeWhereClause</i> script.
Monospace font	VisualAge scripts and other examples of Smalltalk code	<pre>doSomething aNumber aString aNumber := 5 * 10. aString := 'abc'.</pre>
	Text you can enter	For the customer name, type John Doe

Tell us what you think

Please take a few moments to tell us what you think about this book. The only way for us to know if you are satisfied with our books or if we can improve their quality is through feedback from customers like you. There is an online reader's comment form on the VisualAge Smalltalk web page.

Chapter 1. Introduction

VisualAge Smalltalk ObjectExtender is an extensive and powerful persistence framework providing a complete solution for building robust, scalable persistence support for object models. Object models, represented by class hierarchies, are said to be persistent when instances created from these classes can be stored to an external data store such as a relational database.

ObjectExtender enables you to map objects and relationships between objects to information stored in relational databases. It also provides linkages to legacy data on a number of other systems. As a feature of both IBM's VisualAge Smalltalk and VisualAge for Java™ products, ObjectExtender is especially designed for UI-intensive, nested transaction-type applications. This tight integration with the VisualAge family enables you to leverage your current VisualAge investment and expertise.

A rich set of integrated tools make the persistence effort minimal by providing:

- Automated code-generation services for underlying frameworks
- Import and export facilities for working with database schemas
- Import and export tools for VisualAge Smalltalk UML Designer models
- Debug and monitoring tools for performance tuning where desired
- Parts to assist visual programming and the building of a GUI to support the transactional semantics of a nested transaction application

As object-oriented technology has matured in the industry, it has proven to be an excellent solution for modeling problems, building prototypes, and rapidly deploying applications. Though object models for these applications are often reused in other applications, one of the costlier tasks of development has been that of translating between object-oriented and non-object-oriented representations of business models. Given that the majority of databases in use today are non object-oriented, the task of mapping objects to relational database tables and legacy data from various sources has been the missing piece in object persistence standards. On a small scale, object persistence is not difficult to solve. However, large scale applications introduce new requirements to frameworks that support object persistence.

Underlying ObjectExtender is a framework that delivers on requirements that apply to large scale applications. As with any good design, trade-offs and compromises must be made. The better you understand your object model and your data model, the better you will be prepared to take advantage of the powerful function that ObjectExtender offers. ObjectExtender provides this support in the following ways:

- Minimal intrusion to object and database design
- High performance
- Advanced transaction support
- Advanced query support
- Relationship support
- Seamless support for various database paradigms

A brief look at each of these requirements follows.

Minimal intrusion to object and database design

Close coupling between the two models, object and database, is history. Object models only need to represent their application domain, not the database design. This enables you to maintain an object-centric view of persistence rather than a data-centric view of data stores. The number of persistence constructs required in the application code is very low, keeping the persistence application model relatively lightweight. Separating objects and database concepts is done by employing meta-information. The loose connection between object model and database is preserved through a rich set of mapping schemes.

ObjectExtender supports this meta-information layer through UML object models, mapping models, and data models (schemas).

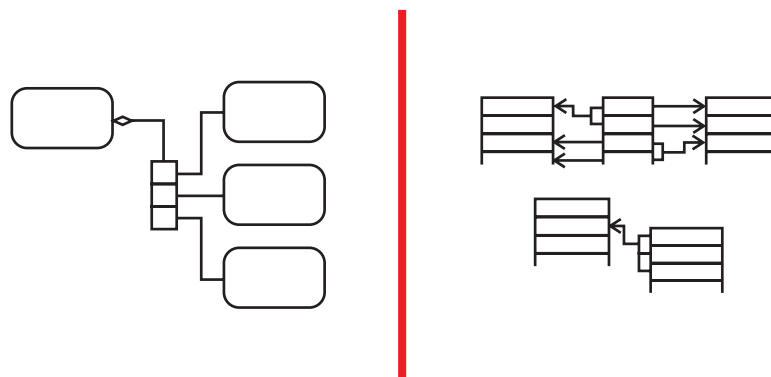


Figure 1. Object model and database design. Using ObjectExtender, you can maintain a distinct boundary between your object model and data store design. Neither design needs to be retrofitted to accommodate the other.

ObjectExtender is integrated with the visual aspects of programming as well such that existing view components can work with framework.

High performance

System performance depends on many factors: server load, network throughput, client application response time, and more. Database access can be optimized according to these performance factors.

To optimize the number of database round trips and path length to data, a cache and preload framework enables you to define your working set of data for any task. This working set determines the amount of data that will be retrieved for a given task.

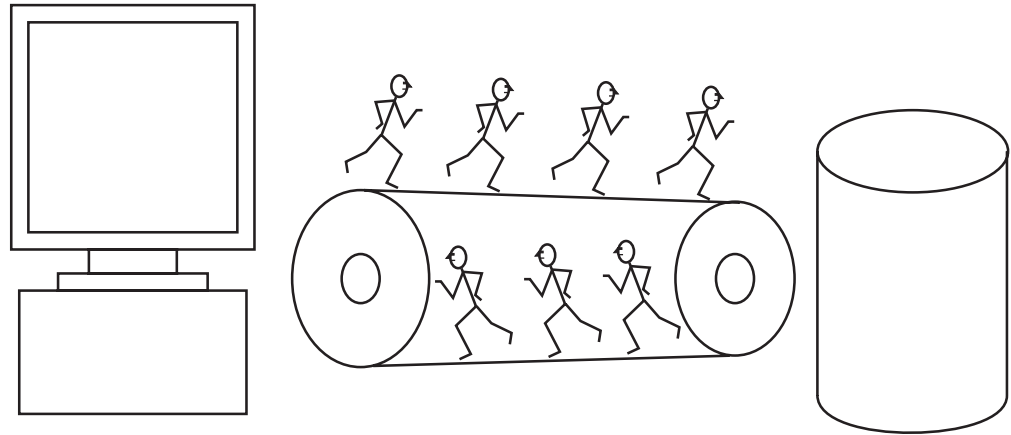


Figure 2. Network round trips are reduced improving performance.

When the set of objects you want to retrieve is very large, performance degradation occurs when instantiating the objects all at once. A cache framework prevents this problem by keeping objects in their native format as long as possible.

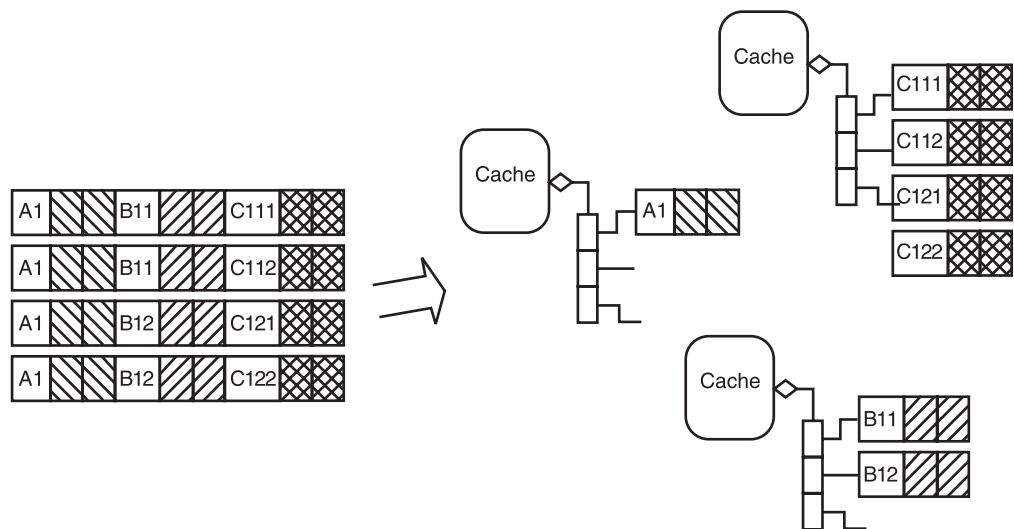


Figure 3. Caching framework. Using a caching strategy optimizes performance.

The set of data required to perform a task is often known ahead of time, or can be derived from application hints. Minimizing the number of database round trips is achieved by retrieving as much data as possible. The preload framework enables you to tune performance in this manner. One example of this would be joining the relational tables that form a composition tree.

In the diagram below, two *Dept* objects are retrieved from the database by the application. Other related objects, *Emp* and *Addr* are also retrieved and loaded into the cache to be instantiated at a later time, such as when accessed by a user-interface dialog. This eliminates the extra trips to the database to fetch the related objects.

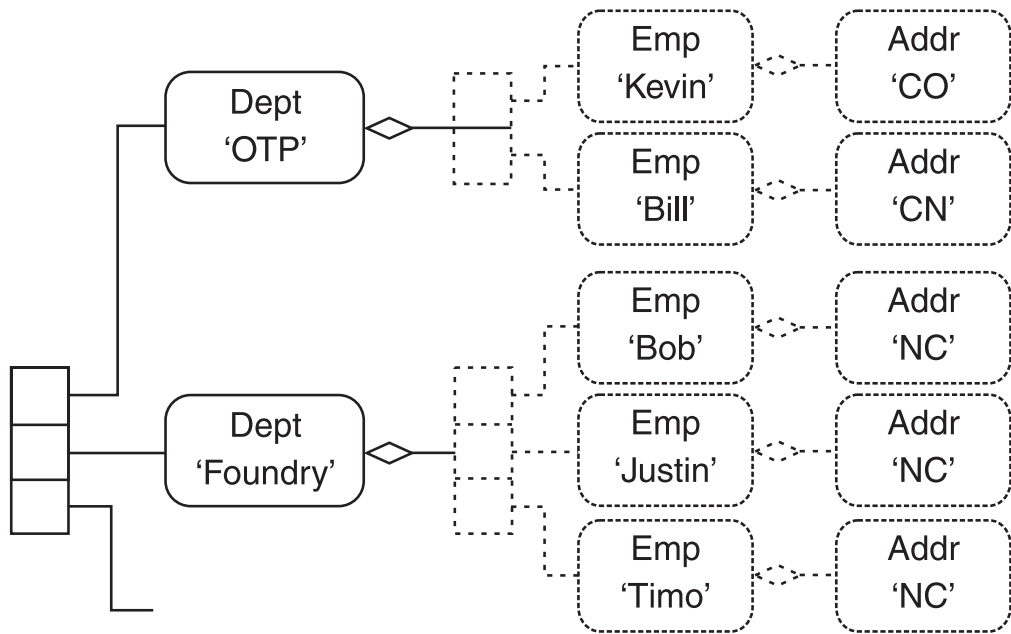


Figure 4. Preload strategy. Loading other objects ahead of time based on derivable data and/or application hints reduces trips to the database.

Support for Static SQL is available as well to reduce server load.

Advanced transaction support

The transaction framework enables you to support multiple users executing concurrent and nested transactions that update the same objects. You can manage multiple versions of objects in your image, and determine appropriate policies for handling collisions and commits to the database.

The transaction framework treats the application and database memory spaces as if they were one memory space; synchronizing them and managing collisions.

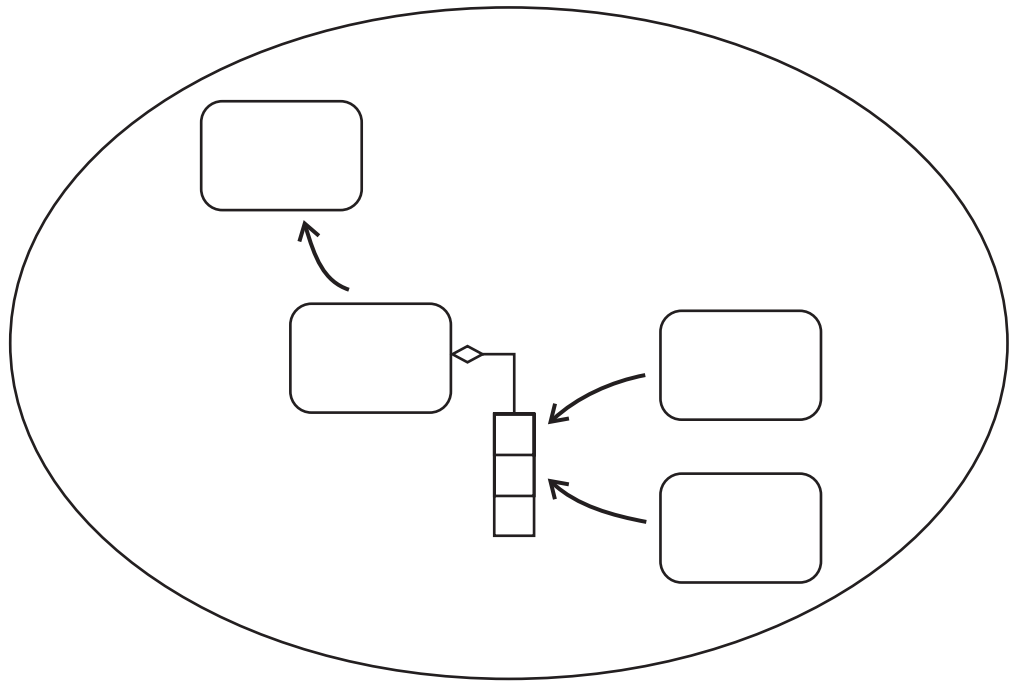


Figure 5. Application and database memory spaces. The transaction framework enables you to use one set of semantics that is understood by the database and the object model. This makes managing synchronization between them easier.

Advanced query support

You can define very complex object models and relationships easily using the powerful ObjectExtender tool set. The underlying query model is expressed in object terms rather than on database-specific native terms. Several kinds of mapping strategies encourage this loose coupling.

The code-generation function produces sophisticated SQL statements customized for your object model. The SQL statements are consumed by service classes that manage your data. Because the code-generation is done at development time, you can benefit from the advantages of static SQL or tune them as desired rather than having them generated at runtime.

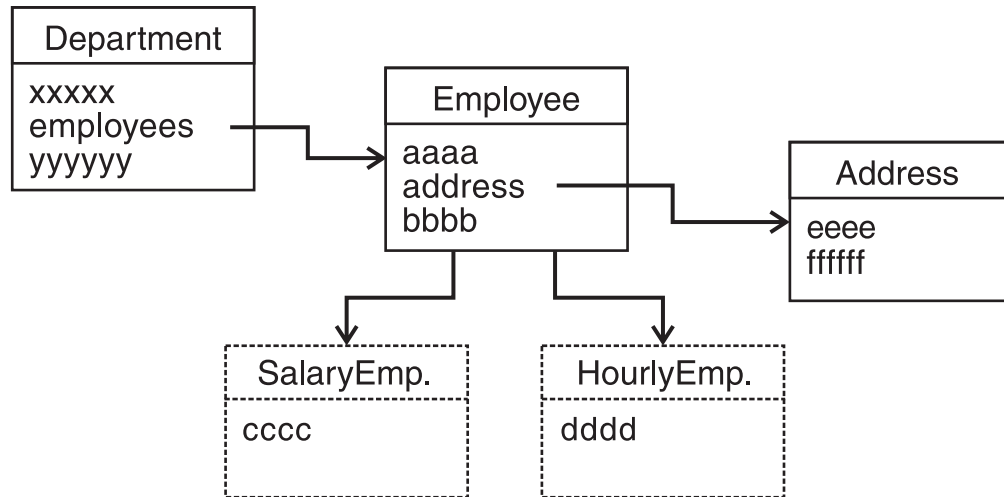


Figure 6. Advanced query capability. Class hierarchy queries are expressed in object terms.

The query generator supports:

- Equi-joins for loading chains of objects (no branches).
- Unions and set differences for loading complex composition trees.
- Left-outer-joins for loading trees that allow missing leaves.

Relationship support

Relationships between persistent objects are implemented using first-class link objects. Links are hidden behind the generated accessors. Links automatically manage:

- Object referential integrity
- Database key referential integrity
- Persistent state of the relationship (lazy retrieval).

Seamless support for various database paradigms

Large scale commercial applications typically need to access data from multiple data stores.

The relational and procedure-call generation embedded in the framework provides the linkages between the object model and the data store. These linkages are established through generated service classes and include generated queries or data access statements.

The service classes execute the generated SQL to map the object model to the following relational data stores:

- DB2
- ORACLE
- ODBC
- JDBC

Table definitions can be read from an existing database schema. The service classes also invoke the generated record mappings to access transactional data in CICS/IMS or other legacy datastores.

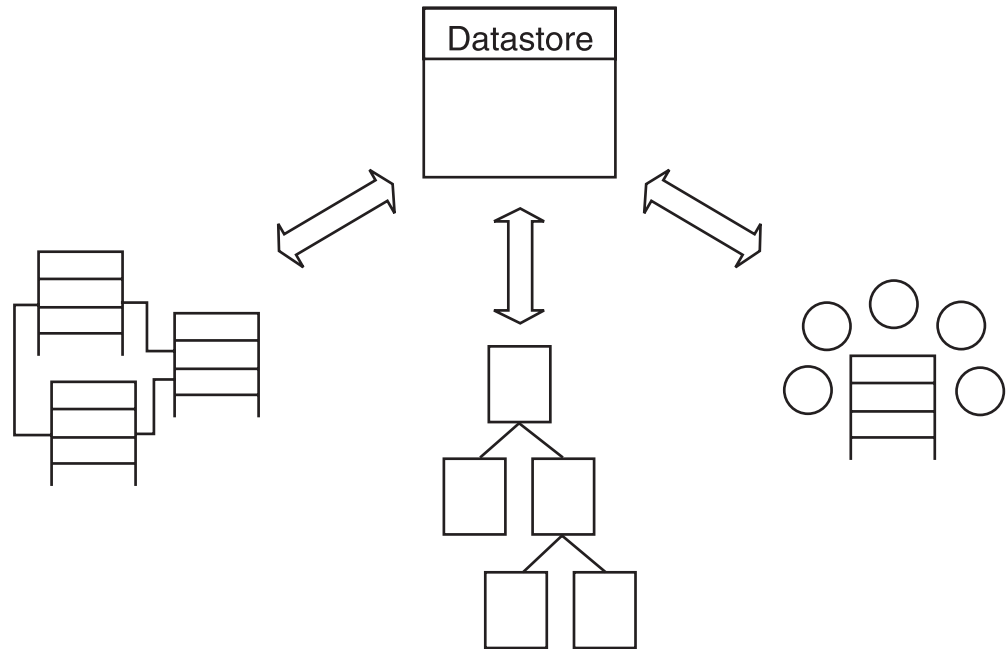


Figure 7. Various database paradigms. Three data stores are shown: relational, hierarchical, and data encapsulated within a service layer. ObjectExtender currently supports relational databases and the encapsulated service layer.

For each business object class there is a set of pluggable data access services. There can be more than one services set per business object class. For example, a relational service set and a procedure call service set. Activation of the data store determines which service set a business object class is currently using.

Chapter 2. Concepts

Key elements managed by the framework

The primary elements managed by ObjectExtender are:

- **Business objects.** Business objects are typically the objects in your application domain that you want to persist in a data store, a bank application's *Customer* and *Account* objects, for example.
- **Relationships.** Objects and relationships are very similar to the entities and relationships of extended entity-relationship modeling. A primary responsibility of ObjectExtender is to maintain the integrity of objects in the run-time image and the corresponding persistent data store through the support of atomic transactions. For example, a *Customer* object could have a one-to-many relationship with a collection of *Account* objects such that each *Customer* knows its set of *Accounts*, and each *Account* knows its *Customer*. The transactions necessary to manage this relationship are transparent to you.
- **Transactions.** Transactions represent paths of code execution. For example, a bank application might have a transaction that updates a customer account. The code activity necessary to manipulate the persistent objects would take place in the transaction.

Key tasks you develop

Building your object models on the ObjectExtender framework can be done in several ways. How you begin depends upon whether your data store already exists. Whether you are starting from scratch, or creating an application to persist to a legacy database, it is best to separate your application domain into three programming tasks:

- Object modeling
- Application programming
- Data access programming

These tasks can be implemented by a team of programmers or by one. In either case, the framework addresses all of the details required for persistent object behavior. For each task, browsers, tools, and code generation services are provided.

Object modeling involves defining the application domain in terms of objects and the relationships between them. ObjectExtender facilitates the task of mapping object models to a range of back-end storage and processing technologies, most notably new and existing relational databases and transaction processing monitors.

Application programming comprises implementing scripts on the model objects, relationships objects and other application objects that implement the functions, business rules and user-interface of the application. ObjectExtender provides an extensible transaction abstraction that gives application programmers a framework to implement the logic of the basic use cases of an application.

Data access programming implements the individual services that map objects and relationships in the object model to the underlying data store using such mechanisms as SQL and transaction processing monitor based calls. For relational databases, ObjectExtender provides a code generation services that create an implementation of a data access service layer from a higher-level mapping specification. The generated code is complete. Enhancement to the generated code

is only needed in special cases, for example, tuning the performance of special SQL queries because of special knowledge of underlying data or application access patterns, or because of unusual table structures in existing database schemas.

Organizing your application

The following diagram shows the organization of the application as seen by ObjectExtender .

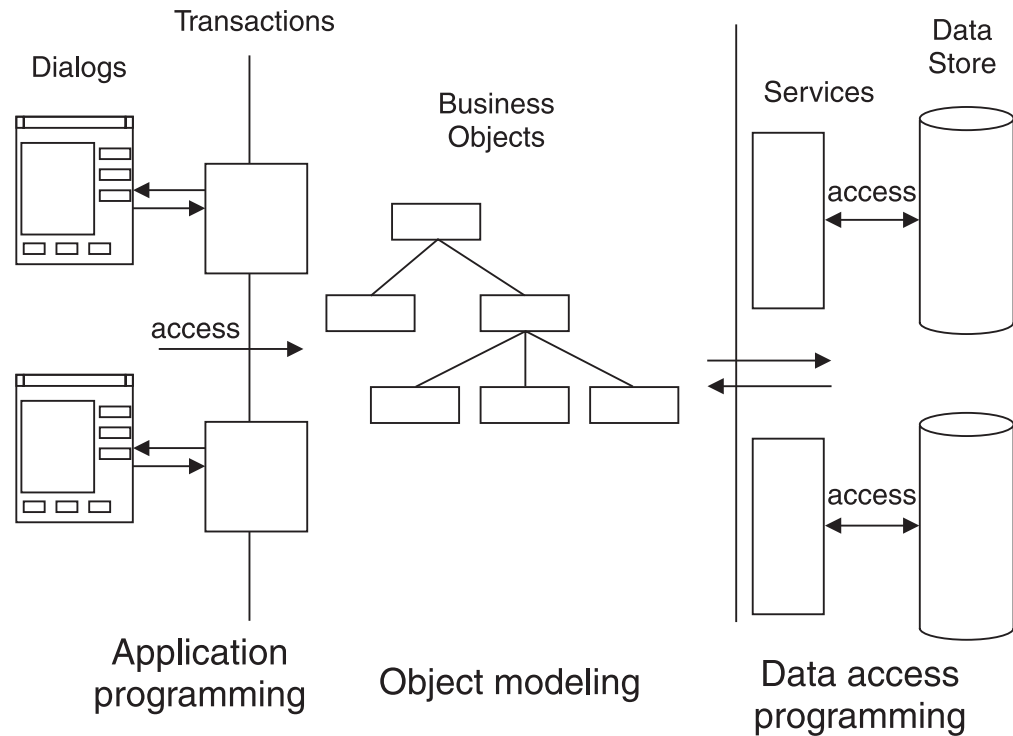


Figure 8. Application organization. Object modeling, data access programming, application programming: these tasks comprise the main elements for using the framework.

ObjectExtender helps you organize your application into three main components in the following ways:

- **Object Model.** Using the Model Browser, you define the basic shape of your persistent objects. In UML terms, you describe their attributes and associations. Code generation services are used for creating the domain classes that will contain the persistence support for your business objects.
- **Transactions.** Transactions represent paths of code execution. Transactions guarantee the consistency of data in the image. You create transactions that access and manipulate persistent objects. Any dialogs necessary for interaction with the application user are implemented by you as well.
- **Services.** Services are classes that implement the low-level functionality required to store and retrieve objects from the data store. Each persistent class has a corresponding service class. Service classes can be generated automatically from meta-information you provide by mapping your object model to a data schema. The Schema Browser is used to define new or import existing database schemas, and the Map Browser is used to map the object model and data schema.

The persistence framework coupled with the robust development tool set provide everything you need to produce a relatively lightweight, industrial strength

runtime framework for any application. As an OO developer, your main concern can be the object model understood as a set of business objects with attributes individually and relationships communally. This understanding is described using the development tools which create meta-information that the persistence framework uses to produce the persistence constructs that support your runtime application.

Figure 1 encapsulates what ObjectExtender offers both in terms of development and runtime environments.

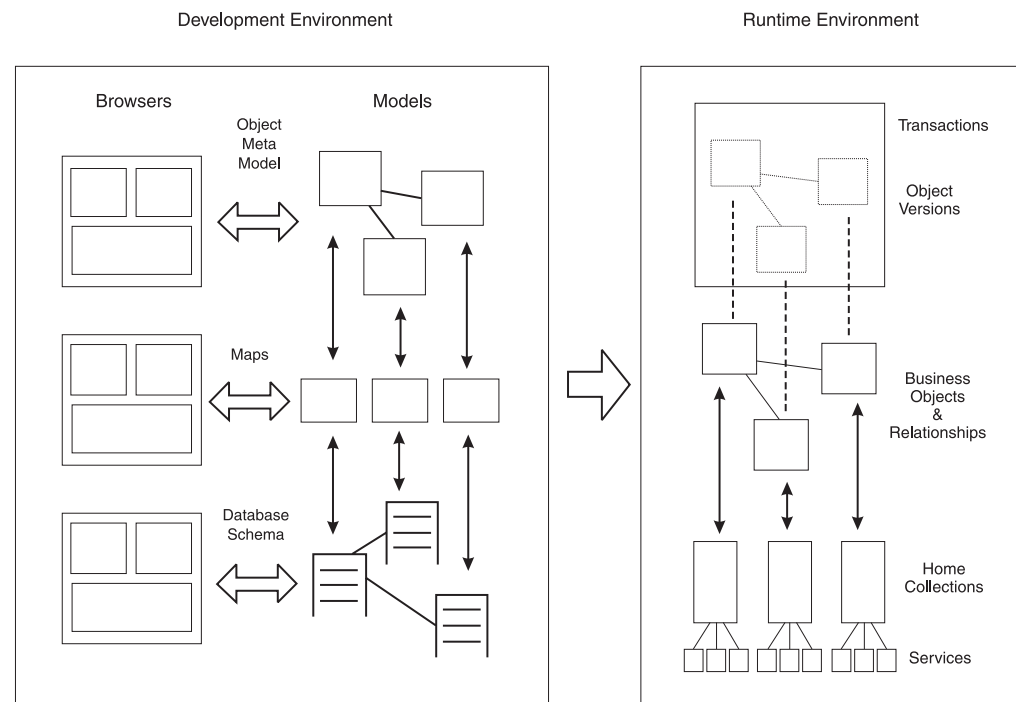


Figure 9. The development and runtime environment perspectives. Implementing your application with ObjectExtender involves using browsers at each level of abstraction during development. Each abstraction describes a particular aspect of your business objects to the transparent framework. Each layer of the framework generates the necessary code and services for the object model to persist in the runtime environment.

The main components of the framework

ObjectExtender provides a runtime system for application developers. It is supported by a rich set of tools that are used to provide input to transparent frameworks which generate the persistent layer for object models. As a developer, you will primarily use the tools, especially, if your object model is to persist in a relational data store. It is helpful, however, to be acquainted with the main frameworks, particularly if you want to create your own data services layer for non-relational data stores, or, if you want to do some performance tuning. Framework highlights are provided here; further details are in the reference sections.

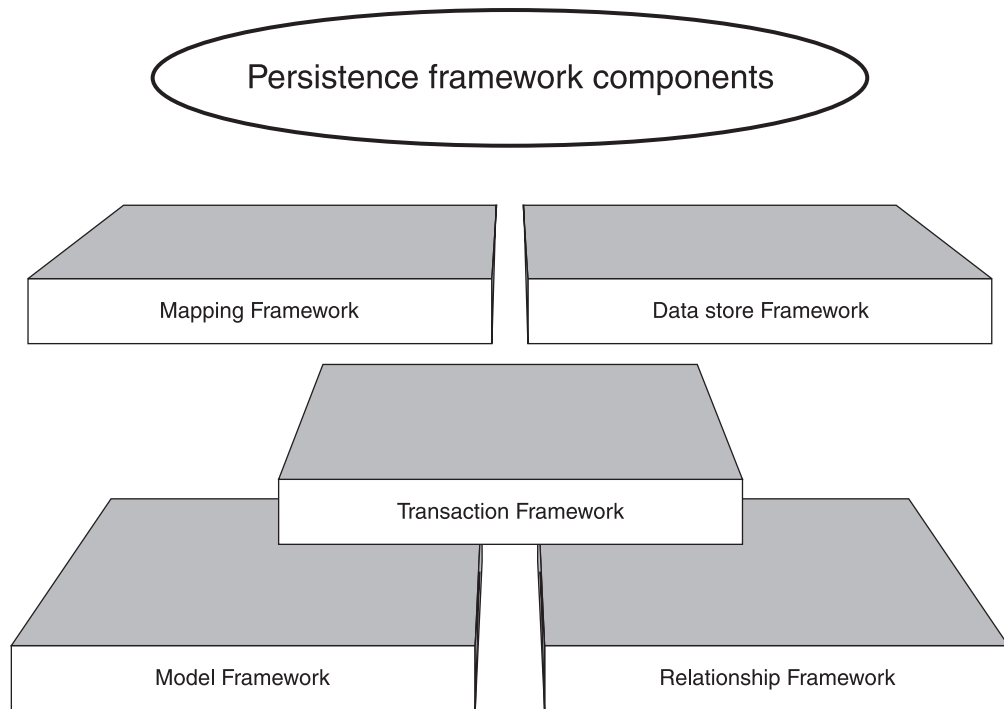


Figure 10. Components of the persistence framework. The subsystems underlying the ObjectExtender feature are designed to address the persistence issues for both large and small scale applications.

The ObjectExtender framework is composed of the following:

- Modeling framework
- Relationship framework
- Transaction framework
- Mapping framework
- Data store framework.

Model framework

The model framework supports describing the state and behavior of persistent objects. The result is the runnable implementation of the object model.

The model framework provides a pure object interface to your application supporting the description of model attributes and associations and implementing accessors to the model data. It supports the concept of defining attributes for uniquely identifying an object, that is, key attributes map to relational keys or key transaction data tying models into relationships and transactions.

Relationship framework

Relationships are a missing feature of OO languages but an essential feature of real-world applications and data. Relationships between persistent objects are implemented using first-class link objects. The link objects manage:

- Object referential integrity
- Database key referential integrity
- Persistent state of the relationship

The relationship framework provides key to object-based user interfaces. It automatically manages the following relationship semantics:

- **Requiredness.** Consider a relationship between a *Person* object and a *Car* object. *Person* has a collection of *Cars*. Requiredness on this relationship might specify that a *Person* requires at least one *Car*.
- **Cardinality.** Cardinality specifies how many objects may participate in the relationship (one-to-one, or one-to-many). For example, the cardinality of the *Person-Car* relationship could be defined as "one-to-many". That is, one *Person* owns a collection of *Cars*.

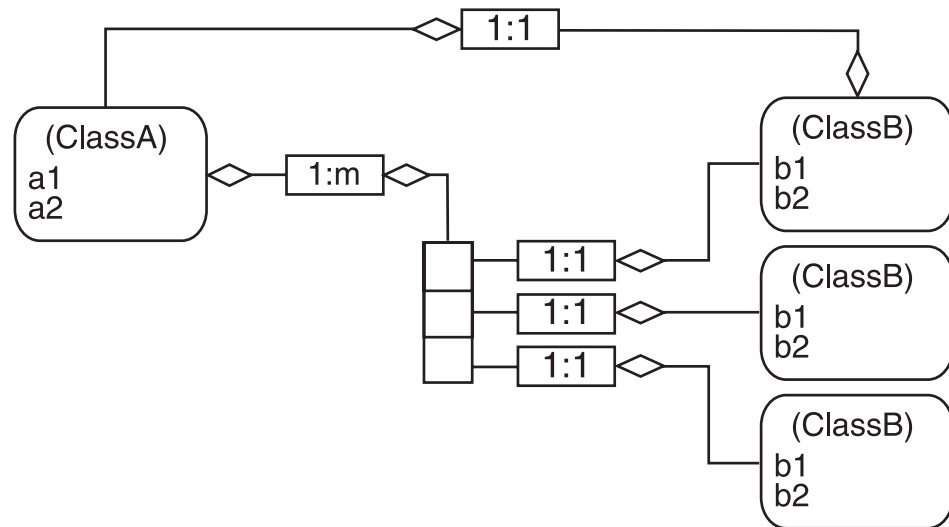


Figure 11. Relationships can be uni-directional and bi-directional. The diagram shows a two-way (bi-directional) relationship: a one-to-many and one-to-one relationship between the two classes.

- **Inverse maintenance.** Continuing with the one-to-many relationship, *Person-Car*, the framework requires that one-to-many relationships be defined as two-way (bi-directional) relationships. Thus, a *Car* object would have a one-to-one relationship with a *Person*. If a *Car* is added to a *Person*'s collection, then the framework ensures that the *Car*'s owner is set to that *Person*. Likewise, if a new *Car* object were instantiated, the framework ensures the addition of the instance to the collection belonging to *Person*.

Transaction framework

Operations on persistent objects have transaction semantics similar to the data stores in which they persist. The transaction framework provides that all operations within a transaction are either committed or rolled back together so that a consistency is maintained within the OO environment. Transactions are introduced into the persistence binding through a set of synchronization and collision management schemes.

Synchronization schemes

Synchronization between application and data memory spaces is achieved by various synchronization schemes. These schemes define when modified objects in the application memory are sent to the database and when objects are refreshed in the database. For example, a deferred write scheme would imply that modified objects are first recorded within a transaction, then, when the transaction is committed, the modified objects are automatically written to the database all at once.

Collision management schemes

Collision management schemes provide different approaches according to the different types of transactions defined. Transactions with a high penalty for failure, for example, could have a pessimistic collision prevention scheme, whereas transactions with a low penalty for failure, that is, it is worth the risk of failure to gain the efficiency, could have an optimistic collision detection scheme.

A flexible collision management scheme is based on the properties of the transaction, the domain class, and the data store. For example, within a transaction there may be participating objects that are not candidates for collision. When the transaction directs its participants to lock their resources, the non-collision candidates will do nothing since they have no resources that require locking.

The net effect of collision management strategies depend entirely on what the underlying data store supports.

Mapping framework

The mapping framework enables you to map flat data to object inheritance hierarchies. It can generate queries to retrieve appropriate data, determine the appropriate object class for data based on discriminating fields, and resolve relationships of abstract classes to existing subclasses. In addition, it implements the conversion between data and objects. This is done by mapping the data schema elements to object attributes. Object relationships can be populated from data relationships (foreign keys) as well. Object queries are converted to data store queries so that the appropriate objects can be retrieved.

Several kinds of mapping strategies are available.

Attribute mapping

Attributes of a class can be mapped to one or more columns in a table as well as across many tables.

Inheritance mapping

Class hierarchies can map to single or multiple tables. Two kinds of partitioning are supported:

- Figure 2 shows typed partitioning. A class hierarchy maps to a single database table.

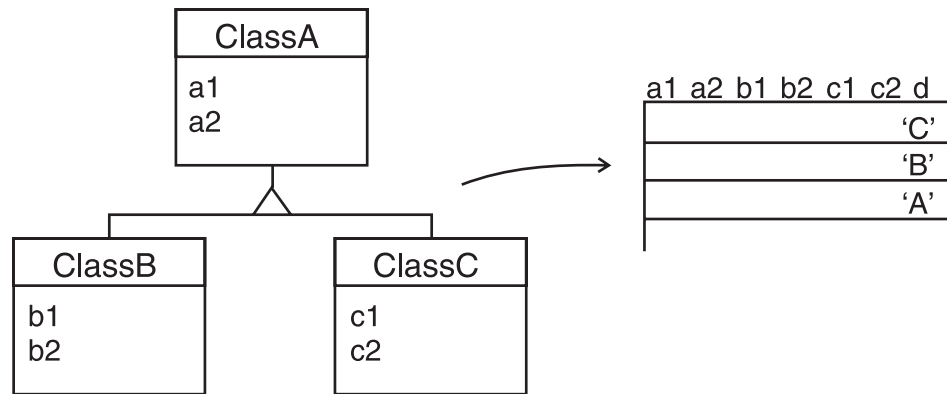


Figure 12. .

- Figure 3 shows vertical partitioning. A class hierarchy maps to root and leaf database tables.

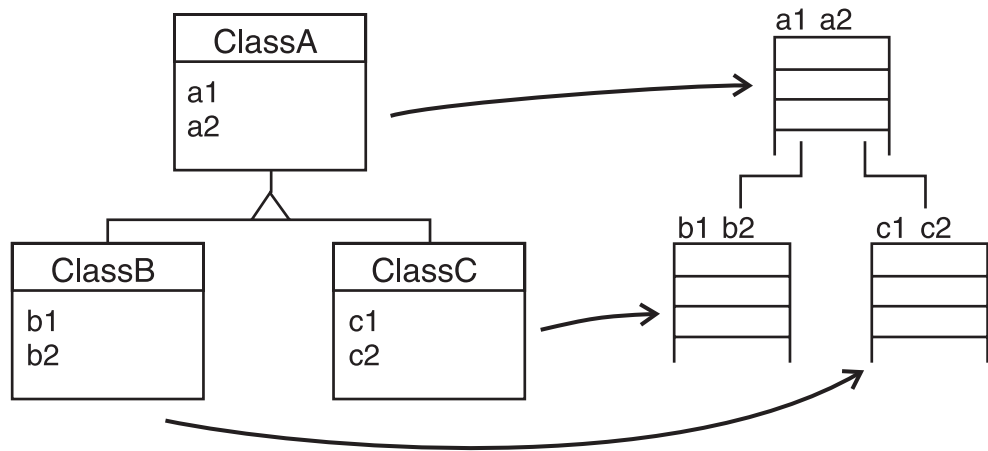


Figure 13. .

Relationship mapping

Objects can be mapped as one-to-one or one-to-many relationships.

- Figures 4 and 5 Show one-to-one relationships. One-to-one relationships can be mapped a couple of ways:
 - Using a backward pointing reference.

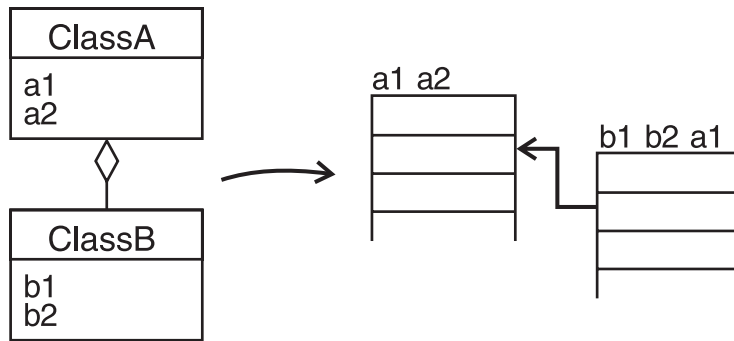


Figure 14. .

- Using a forward pointing reference.

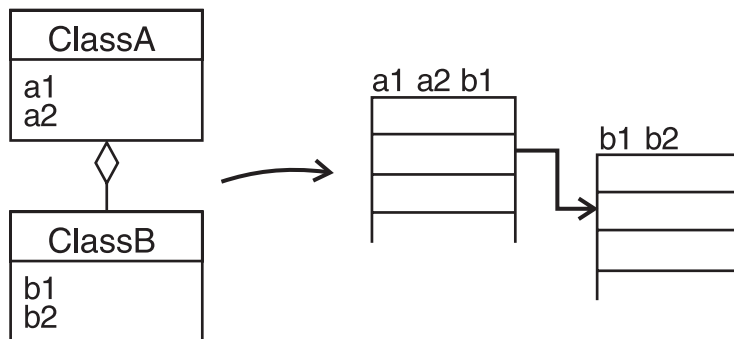


Figure 15. .

- Figure 6 shows one-to-many relationships.

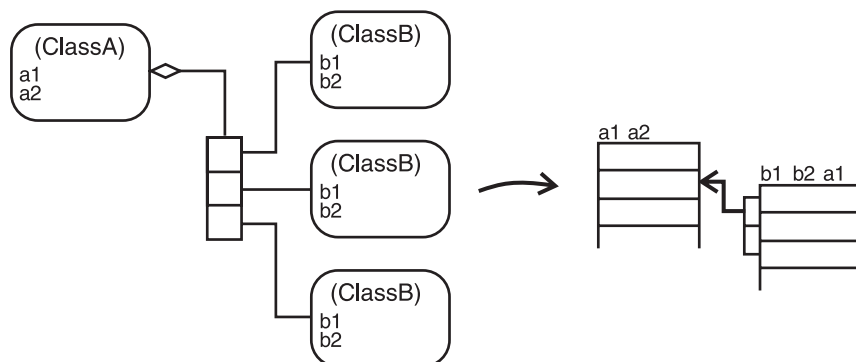


Figure 16. .

Data store framework

The data store framework implements the underlying data store for the persistence system.

This is achieved by capturing a sufficient amount of database meta-information. A logical structure of the data store is specified by the data store schema that represents the structural relationships between schema elements. The logical

structure presents a more user-friendly specification for your object model since often the physical schema of data stores have naming restrictions and other constraints.

The data store framework presents interfaces for

- Connecting to a data store
- Executing database queries
- Managing transactions

Multiple data store types may be used in a single application.

A data store schema can be defined from scratch or imported from an existing database.

Chapter 3. Quick tour

A quick tour of ObjectExtender follows.

Choosing an approach to persistence

There are different ways to begin using ObjectExtender:

- **From scratch.** In this scenario, you would define your model (or import it from VisualAge UML Designer), define your schema (or generate one from the model), define the data store map, and then use the code generation services to create the domain classes and services.
- **From legacy database.** In this scenario, a legacy database exists but no object model. You could import the schema from the existing database, define (or import) your model (or generate one from the schema), define the data store map, then use the code generation services as noted earlier.

Shortcuts: As mentioned, from an existing model, you can generate a default schema and vice versa. Generating models from schemas, and schemas from models is accomplished by using the ObjectExtender tools. The Model Browser has a menu option for generating schemas, and the Schema Browser has one for generating models. Consider this: if you have an existing database, you could import your schema, generate your model from your schema, define the data store map, and then use the code generation services.

The following discussion assumes you are using ObjectExtender to build an object model from scratch. That is, you are defining the model, schema, and data store map, and then using the code generation services. Rather than a concrete example to follow, these highlights are intended to help you understand the overall process for using ObjectExtender.

Creating the data services layer automatically or manually

Before we begin, a first consideration is how the data services layer will be built. The data services layer is created using code generation services after you define a model, schema, and data store map. You can use the code generations services to create a complete set of data services when persisting to relational databases. The code is complete and ready to run. You can fine tune it if necessary. If you are not persisting to a relational data store, however, then you need to write your own data services, but not completely from scratch. You can use the code generation services to create code stubs which you can then complete according to the requirements of your data store.

Using code generation services

The three main elements that are defined to support ObjectExtender applications are the model, the schema and the data store map. The information, sometimes referred to as metadata, for each of these is stored in a class in the repository. When a model, schema or map is first created it can be edited without requiring it to be saved. In this case all changes are recorded in the image and are only available to you in the image. To save the work and make it available to other users, use the Save option from each of the browsers. This will store the details of

the model, map or schema into a class in the repository. If a model, map or schema requires saving the browser will indicate this in the text pane on the lower half of the browser.

After a model, map or schema has been saved it can be loaded by other users into their image by loading the storage class the same way any other class is loaded from the repository. (Note that to load a class it must either be a version or else if it is still an edition you must be the developer of the edition).

The browsers are not automatically refreshed when you load the metadata information for a model, schema, or map. This is done with the load available menu option. Selecting this option will hydrate the models, maps or schema in the image from the list of loaded classes that are holding the metadata information. As modifications are made to a model, schema, or map, the changes are recorded locally in the image. The browser will indicate that the class is out of sync with the image by indicating that it is dirty and requires saving. If you decide you do not want to keep your modifications you can revert the model, map or schema to the version last saved to its storage class by selecting the revert menu option. Once you are ready to store your work into the repository the Save option will take the definition of the model, map or schema from your image (which is always the ones the browsers display) and store it into the repository. From here the class can be versioned and then loaded and modified by another user.

It is good practice to version the class containing the metadata for the model, map or schema before another user loads it into their image. Once the storage class has been versioned it can be moved between repositories using the import and export menu options. When the class is exported it takes the metadata with it allowing ObjectExtender models, maps and schemas to be moved between repositories.

When creating a new object model using the ObjectExtender tools, the steps are as follows:

1. Define the model. (You define a model directly in the Model Browser or import a model developed with VisualAge UML Designer.)
2. Define the schema.
3. Define the data store map.
4. Generate the persistence support code using code generation services.
5. Perform any desired tuning.

Defining the model with the Model Browser

In brief, use the Model Browser to define your object model. Your object model is a collection of business objects represented in terms of classes and associations. Once defined, the persistence support for your model is created by using code generation services.

To create a new object model, first determine the business objects in your model that will persist to the data store, then create the classes and associations that will represent your business objects.

Creating models. To create a model:

1. Launch the Model Browser.
2. Select **New Model** from the **Models** menu.
3. Name the model.
4. Click **OK**.

Creating classes. Continue defining the model by creating the classes and associations that represent the business objects for the model. To create a class:

1. In the **Models** view, select the model name.
2. From the **Classes** menu, select **New Class**. This launches the Class Editor.
3. Provide a unique name for the class. Make sure that the name is unique not only for your model but also for the classes loaded in your image.
4. Next, define attributes for the class:
 - a. In the **Attributes** tabbed dialog box, click **New**. This launches the Attribute Editor.
 - b. Provide an attribute name.
 - c. Select an attribute type.
 - d. Click **Value required**. This indicates whether your model requires a value for this attribute or not.
 - e. In the Attribute Editor, click **OK**.
5. Define the object identifier by selecting one or more of the defined attributes and moving them to the **Object ID** view. To move an attribute in this view, click on the double arrow (>>).
6. In the Class Editor, click **OK**.

Creating associations. To create an association between two classes:

1. From the **Associations** menu, select **New Association**.
2. Provide an association name.
3. Select two classes in the model that will have an association.
4. Each class will play a role for the other class. For each class, do the following:
 - a. Type a name for the role that the class will perform for the other class.
 - b. Specify whether the role is navigable from the other class in the association.
 - c. Specify the cardinality of the role. Cardinality is expressed by specifying the **Many**, and **Required** choices in some combination.

Generating business objects. After you have completed the model description, generate the domain classes using the code generation services.

- From the **Models** menu, select **Generate**, and then select the appropriate options from the SmartGuide.

What the code generation services create. The generation of the model will produce the following:

- *Business object* classes with accessors for all defined attributes and navigable relationships.
- *Home collection* classes, which implement the instance creation, look up by key, and *allInstances* support for a *business object* class.
- *Key* classes which look up *business object* instances using keys.
- *Relationship* classes, which manage relationships between *business objects*.

Once the *business object* classes are generated, you can create transactional instances of the classes and manipulate them and their relationships in transactions. Refer to “Managing business objects” on page 111.

Saving models. Save your model definition to an application and storage class that you supply. Saving your model enables you to take advantage of the existing library management functions. These functions are the same ones you use when sharing code with other developers such as versioning, releasing, and loading different editions of an application.

Suggestion: A useful naming convention is to name your application as follows: *XYZMetadataApp* where the XYZ is some prefix you choose. Use this application to store a model, schema, and map for a given application.

Recommendation: Save the model definition to an application different from the generated model classes. The model will generate into any previously generated schema that has the same name as the one it would have created. For existing columns, it does not modify information such as field type or length. If attributes or roles are deleted, the previously generated columns are not dropped and this is managed by the user.

Importing a model from UML Designer

If you use the VisualAge UML Designer feature to model your application, you can then import model elements from UML Designer into the ObjectExtender Model Browser. To import from UML Designer, follow these steps:

1. In the Model Browser, select the model you want to contain the imported model classes.
2. Select **Import From UML Designer** from the **Models** menu (or the pop-up menu).
A window appears listing all available UML Designer models.
3. Select the UML Designer model that contains the elements you want to import into ObjectExtender. Then select **OK**.
A window appears listing all of the UML Designer class designs in the selected model.
4. Select the UML Designer class designs you want to import. Then select **OK**.

When the import operation completes, the selected ObjectExtender model contains a model class for each of the imported UML Designer class designs. Information from UML Designer is mapped to ObjectExtender as follows:

- The attributes of each UML Designer class design (and of any protocols it conforms to) become attributes of the corresponding model class. An attribute is not created in the ObjectExtender model if the UML Designer attribute's type protocol does not have a conforming instance class design.
- If the class design has an attribute using the Key Attribute idiom, ObjectExtender generates an oid for the corresponding model class.
- Any associations between class designs are preserved as associations between model classes, provided both source and target classes are imported for each association.
- Inheritance relationships among class designs are preserved in the imported model classes.

Defining the schema

In brief, use the Schema Browser to define a schema. The Schema Browser handles descriptions of tables, their columns, and key (primary and foreign) definitions.

Creating schemas. To create a schema:

1. Launch the Schema Browser.
2. From the **Schemas** menu, select **New Schema**.
3. Name the schema.
4. Click **OK**.

Creating tables and columns. To create a table for your schema:

1. Select the schema name in the **Schemas** view.
2. From the **Tables** menu, select **New Table**. This launches the Table Editor.
3. Provide a logical table name. This is a logical name. It does not store in the database.

4. Provide a physical table name. This name stores in the database.
 5. Create the columns for the table:
 - a. In the **Columns** view, click **New**. This launches the Column Editor.
 - b. Provide a column name. This is a logical name.
 - c. Provide a physical column name. This name stores in the database.
 - d. Choose a **Type** for the column.
 - e. Under **Type details**, choose a **Converter** for the column. (Converters translate data. For example, a CHAR datum in a database converts to a *Boolean* object using the *VapCharToBoolean* converter).
 - f. Click **Allow nulls** accordingly.
 - g. In the Column Editor, click **OK**.
- Select a column(s) to be the **Primary key** and click the double arrow (>>).
6. In the Table Editor, click **OK**.

Using Converters. A converter is used to change a class attribute type into another type before storing to the database. When reading from the database, the converter will change a particular database type into the correct class attribute type. For example, if you have a class attribute that has a Date type and the database column has a SQL type of DATE, no converter is needed. However, if you wanted to take this same class attribute and store it in a database column of SQL type VARCHAR, a converter is necessary because the class attribute of Date does not map directly to this SQL type.

The default converter used for fixed character columns is the *VapTrimStringConverter*. This converter truncates spaces from the end of the column value. If an object's identifier contains this column, object cache lookup is done with the truncated value, thus eliminating the chance of false cache hits or multiple cache entries. The drawback of this scenario is the key derived from a relating object will have a truncated value, and the resulting query will not find the appropriate rows in the database. Changing the converter to *VapConverter* will solve the later problem, but you must be aware of trailing spaces that become significant in object lookups.

Exporting schemas. Once your schema is defined, and saved, you will export the schema to the database. Note that the column name used for constraints uses the physical name.

You can wait to do this step until you are really ready to use the database. This is done as follows:

1. From the **Schemas** menu, select the **Import / Export Schema** menu, then select **Export Entire Schema to Database**.
2. Fill in the **Database Connection Info** dialog.
3. Click **OK**.

Saving schemas. Save your schema definition to an application and storage class that you supply. Saving your schema enables you to take advantage of the existing library management functions. These functions are the same ones you use when sharing code with other developers such as versioning, releasing, and loading different editions of an application.

Suggestion: A useful naming convention is to name your application as follows: *XYZMetadataApp* where the XYZ is some prefix you choose. Use this application to store a model, schema, and map for a given application.

Recommendation: Save the schema definition to an application different from the generated model classes.

Defining a data store map

To do this step, you need a model and a schema.

In brief, use the Map Browser to define a data store map. A data store map logically connects the object model description with the data store schema description. This is used for code generation to determine the proper SQL and supporting code needed for persisting the model objects. Using the Map Browser you create table maps for your business objects. All of the table maps collectively comprise your data store map.

Creating a table map consists of mapping the model attributes and relationships to the schema. For each model attribute, you specify the column which should be read to populate the attribute. For each relationship, you specify the foreign key relationship which is its persistent representation.

When all of the table maps are defined, the persistence support for your data store can then be created using code generation services.

Creating maps. To create a new data store map:

1. Launch the Map Browser.
2. From the **Datastore_Maps** menu, select **New Datastore Map**.
3. Provide a data store map name.
4. Select the names of the model and schema that you are going to map.
5. Click **OK**.

Creating table maps. A class must have at least one table map. The table map specifies the database table to which the class will be mapped. Table maps contain property and relationship maps which refer to columns and keys respectively. To create a table map:

1. From the **Table_Maps** menu, select a table map. There are several kinds of table maps.
2. Select the name of the database table you are mapping.
3. Click **OK**.

Creating property maps. Create property maps for each table map. To create a property map:

1. From the **Table Maps** view, select the table map.
2. From the **Table_Maps** menu, select **Edit Property Maps**. This launches the Property Map Editor.
3. **Mapping attributes.** In the **Attributes** tabbed dialog box, map the class attributes to table columns by selecting the attribute, the type of map, and the table column.
4. **Mapping relationships.** In the **Association Roles** tabbed dialog box, map the class associations to foreign key relationships.
5. Click **OK**.

Generating services. When you have completed all of the table maps, generate the data services using the code generation services.

- From the **Datastore_Maps** menu, select **Generate Services**, and then select the appropriate options from the SmartGuide.

What the code generation services create. The code generation services will create the following code:

- A *DataStore* class, which manages all the service classes for this data store map, and manages the pool of database connections.
- A *ServiceObject* class which implements the appropriate services for each model class, and is supported by:
 - A *QueryPool* class containing the query specs for the basic services.
 - *Injector* classes to organize query inputs.
 - *Extractor* classes to organize the reading of query results into the appropriate caches and data objects.
 - A *DataObject* class for each business object class, which holds the raw data for a business object and provides cache entry behavior and access to key values.

Saving data store maps. Save your data store map to an application and storage class that you supply. Saving your map enables you to take advantage of the existing library management functions. These functions are the same ones you use when sharing code with other developers such as versioning, releasing, and loading different editions of an application.

Suggestion: A useful naming convention is to name your application as follows: *XYZMetadataApp* where the *XYZ* is some prefix you choose. Use this application to store a model, schema, and map for a given application.

Tuning the code as needed

If desired, you can now perform any tuning or installation specific changes to the generated code. These might include SQL editing, service caching improvements, and more complex installation specific OID generation for example.

For example, suppose you need to a specific query ordered by a particular column in a table. You could create the script *myTableOrderedBySomeColumn* in the *XYZYourClassQueryPool* in the *sql services* protocol.

Always ensure that your altered set of service classes still support the full required service protocols.

Completing your application

Having defined your model, schema, and map, and having generated the persistent code layer for the model and data store, you can now begin building the next layer of your application, a GUI layer for example. Using the Composition Editor, you can use the parts in the ObjectExtender palette category which will make your business objects transaction aware.

Exporting your model to UML Designer

Once you have defined your model with ObjectExtender, you can also export it to UML Designer for further modeling and analysis.

To export your model to UML Designer, follow these steps:

1. In the Model Browser, select the model containing the model classes you want to export to UML Designer.
2. Select **Export To UML Designer** from the **Models** menu (or the pop-up menu). A window appears listing all of the available UML Designer models.
3. If you want the exported elements to be placed in an existing model, select the target model and then select **OK**.

If you want to create a new UML Designer model for the exported elements, select **<New>** and then select **OK**. You can then specify the name of the new model, which will be created for you.

4. When prompted, select the ObjectExtender model classes you want to export. (Remember that if you want to preserve associations between classes, you must export both the source and destination classes.) Then select **OK**.

When the export operation completes, the target UML Designer model contains a class design and protocol for each exported ObjectExtender model class. See the *VisualAge Smalltalk UML Designer Guide* for more information about model elements exported from ObjectExtender.

Chapter 4. Tools

The ObjectExtender tools are a collection of development browsers, tools, and code generation services that help you build persistence support for your application. They are available from the **ObjectExtender Tools** option under the Transcript menu.

The browsers are your main resources for building your application on the ObjectExtender framework.

The browsers are used to describe your object model, database schema, and data store mapping, as well as generate the code the framework needs to manage your business objects. Using the Status and SQL Query tools, you can collect and study information to ensure that your business objects are exhibiting the behavior you expect.

The browsers and tools are briefly outlined below:

- **Model Browser.** For defining your object model, its classes and associations and generating schemas from a defined model.
- **Schema Browser.** For defining a logical description of your data store to which your object model will persist. Existing database schemas can be imported, and models can be generated from schemas.
- **Map Browser.** For mapping your object model, or persistent classes, to your logical (or database) schema. Each persistent class needs a map which associates the attributes of the class with their corresponding columns (or fields) in the database tables (or record) and also associates class associations with table connections. You must first define your object model and schema before you can map them.
- **SQL Query Tool.** For occasional use, when you need to query the database directly. This tool is great for testing the generated query statements.
- **Status Tool.** For collecting various kinds of statistics during development, such as looking at persistent object, data store, and transaction statistics.

The browsers do not allow any of the meta model entities to be modified if the storage entity is an edition and its developer is not the current image user.

The Model Browser

The Model Browser is used to define the classes in an object model. The object model consists of the classes that represent the business objects in your application that will persist in a data store.

- Browser use
- Browser description
- Browser menu-bar choices

Browser use

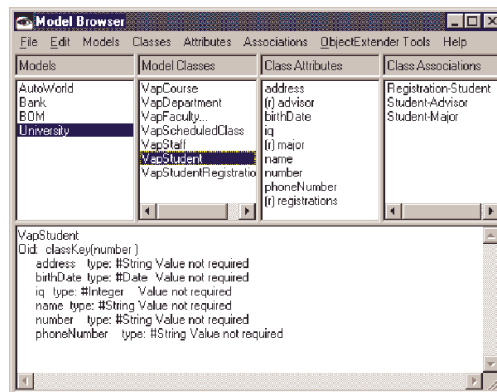
Business objects are described through class descriptions. For example, in an object model for a University you might describe: a *Student* class, a *Faculty* class, a *Course* class, a *Schedule* class, and so on.

Describing a class with the Model Browser is similar to defining a class in a Smalltalk browser. Each class consists of attributes (instance variables) and relationships. Attributes typically describe the granularity of the objects you wish to persist. Relationships describe how objects in your model are associated with one another. In the University model, your *Student* and *Faculty* classes might have attributes such as **studentNumber** and **facultyNumber**, respectively. In addition, there might be a relationship between the two classes, a relationship between Faculty and Student, for example. A Faculty object, for example, could have an **advisedStudents** relationship, or role, with several Student objects.

There is a difference between describing a class in the Model Browser and defining a class using other tools. Describing a class does not create an instance of the class. It creates metadata for the class. The metadata is used later to generate the class instance. The instance of the class is created when you use the **Generate** function under the **Models** menu. Generating the code for your model is typically done after you have described all the classes. If you need to make changes to your model after you have generated the code, you need to generate the code again.

Browser description

The Model Browser is described below.



The Model Browser presents several views:

- **Models.** Displays the names of your models.
- **Model Classes.** Displays the names of the classes for a selected model.
- **Attributes.** Displays the names of the attributes for a selected class.
- **Class Associations.** Displays the associations or relationships defined between the classes in the selected model.
- **Information.** This view is not labeled as such, but it is a read-only view that provides descriptive information in a given context. For example, if you select only the name of a given model, it will provide certain statistics about a model.

Each view is used successively to describe your model, its classes, and its associations. It is easy to verify your design when complete by browsing the contents and associations for each class. When your model definition is complete, you can save it in the library. Models are stored as versions of the class *ModelStorageClass* in the application *VapMetadataStorage*.

In addition to creating models with the Model Browser, you can also edit models, and load other models into your image from the library. You can also use the Model Browser to import and export model information between ObjectExtender and UML Designer.

After you have designed your object model, you describe it using the Model Browser, and then generate the code for it.

Creating a model involves giving it a name and then creating its classes and associations. Creating classes for the model involves giving each class a name, attributes, relationships, and key properties. Creating associations for the model involves giving each association a name and defining the classes involved in the association as well as naming the role each class plays in the association.

You create a model by selecting **New Model** from the **Models** menu, and supplying a name for the model.

You create a class in your model by selecting **New Class** from the **Classes** menu. This launches the Class Editor where you fill out the characteristics of the class such as class name, superclass name, class attributes, or instance variables, relationships the class might have with other classes, and key properties which will uniquely identify the class from other classes.

Browser menu-bar choices

Most of the menu-bar choices for the Model Browser are described below.

- **Models.**

The following choices are available.

New Model

Start here when creating a new model. A dialog prompts you to provide a model name. If you have an existing database there is a shortcut for creating a new model. Using the Schema Browser you could import the schema, then you could generate the model from the schema.

Delete Model

When you delete models, you are deleting them from your browser, not the image.

Save Model

For saving model descriptions. You are prompted for an application and a storage class name.

If you edit a model for which you have already generated domain classes, *remember* to regenerate the code again for the model to pick up your changes.

Load Available Models

Loads all the available models from the storage classes in your image.

Revert Selected Model

Reverts back to the previously saved version of the model.

Model Code Generation Options

Launches a dialog in which you can specify the following default behavior when you generate the model.

- **Model application name**
- **Generate persistent classes**
- **Generate VisualAge parts**

- **Default persistent class root**

Generate

Launches a dialog prompting you for an application name into which your generated code will be stored.

Generate Schema From Model

Generates a database schema from the selected model. This is a shortcut when you are creating an application from scratch. In like manner, you can generate a model from an existing database schema using the Schema Browser .

Import From Modeller

Imports class designs from VisualAge UML Designer into the selected model as model classes. You can select which class designs you want to import. Attributes, inheritance, and associations are preserved (provided both source and target classes are imported for each association).

Export To Modeller

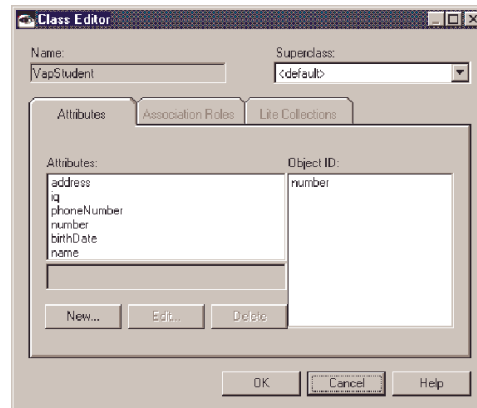
Exports model classes in the selected model to VisualAge UML Designer as class designs and protocols. You can select which model classes you want to export. Attributes, inheritance, and associations are preserved (provided both source and target classes are exported for each association).

• Classes.

Assuming you have created a model, you can then create classes for it. This menu provides the choices for creating, editing, and deleting the classes for your model. The choices are as follows:

New Class

Launches a Class Editor. Be sure to have the model selected first when you select this.



Fill in the following fields to complete your class description:

- **Name.** The class name must be unique.
- **Superclass.** Provide the name of the superclass or use the default.
- **Attributes.** This tabbed dialog box is where you define your instance variables. To define a new attribute, select **New**. This launches the Attribute Editor where you fill in the attribute name and the attribute type, and whether the attribute requires a value.
- **Association Roles.** This tabbed dialog box gives you visibility to all of the roles that have been defined for the class. Class roles are defined

when you create Associations using the Association Editor. Be sure to have the model selected for which you want to display and define associations.

- **Lite Collections.** This tabbed dialog box enables you to define a subset of attributes that you wish to collect and use in your application in some way. To create a lite collection, select **New**, and type a name. The name will appear in the list of Lite collections, and the class attributes (properties) will display in the Class properties view. Next, choose the class properties to include in the collection and click **Apply**. Filter and packeting are optional. If you choose packeting, this collection **MUST** be used with the packeting protocol, either manually or with the packeting container.
- **Object ID** Specify one or more attributes to be the identifying attributes for the class. Select an attribute and click >> to add it as an object identifier.

- **Attributes.**

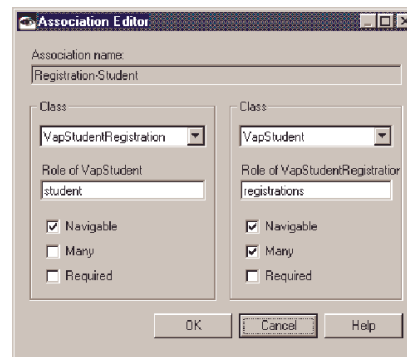
New Attribute

Launches the Attribute Editor. You can use this to create, edit, or delete attributes for a selected class. Attributes typically describe the granularity of the objects you wish to persist. Relationships describe how objects in your model are associated with one another. In the University model, your *Student* and *Faculty* classes might have attributes such as **studentNumber** and **facultyNumber**, respectively.

- **Associations** You can create, edit, and delete associations between classes in your object model using this option. Be sure to have the model selected.

New Association

Launches the Association Editor. Use this option to define a new association between two classes in your object model



You must fill in the following to create the association between the classes.

- **Associated classes.** Supply a name for the association that has semantic meaning for your model. For example, an association name of **Student-Advisor** indicates a relationship where one object is a Student, the other object is an Advisor. This naming convention also hints at the roles that each object will play in the relationship. You must supply the names of the two classes that are in the association. In this example, *VapStudent*, and *VapFaculty*.
- **Associated between classes.** In an association, each class plays a role, that is, each class plays a role in or for the other class.

In the **Student-Advisor** association, the *VapFaculty* plays the role of **advisor** for *VapStudent*. The only role that *VapStudent* will play in the *VapFaculty* class is that of **advisedStudents**, one of many advised students.

The last part of definition for the association has to do with navigability and cardinality (how many).

- **Navigable**. Marking the object navigable means that the role is reachable from its counterpart class in the association. For example, in the **Student-Advisor** association, checking the **advisor** role would mean that *VapStudent* could ask the *VapFaculty* instance for its **advisor**.
- **Many**. Specifying cardinality is done with this choice and the **Required** choice.

You can specify the cardinality of the roles between the two classes as follows:

- **Required** unchecked, **Many** unchecked. This defines a zero-to-one (0:1) cardinality for the role in the association that is played by this object.

When **Required** is unchecked it means that zero instances of this object is valid in the association. When **Many** is unchecked, it means that only one instance is allowed for the object to fulfill its role in the association.

- **Required** checked, **Many** unchecked. This defines a one-to-one (1:1) cardinality for the role.
- **Required** unchecked, **Many** checked. This defines zero-to-many (0:many) cardinality for the role.
- **Required** checked, **Many** checked. This defines a one-to-many (1:many) cardinality for the role.

In the illustration of the **Student-Advisor** association, the **advisor** role played by *VapFaculty* could have a **0:1** cardinality in the *VapStudent* object meaning that the *VapStudent* may or may not have an **advisor**. The cardinality of the **advisedStudents** role played by the *VapStudent* could be **0:many** meaning that the *VapFaculty* object may have no students to advise or many.

The Schema Browser

The Schema Browser is used to describe the schema of the relational database into which the persistent classes are stored. A physical database with tables (for a relational database) is required to support the object model. If the application was developed completely from scratch, the ObjectExtender schema generation of the Schema Browser with tools, could be used to create a schema. ObjectExtender creates the necessary data definition language (DDL) for the database or creates the tables, indices, and constraints directly from the schema.

- Browser use
- Browser description
- Browser menu-bar choices

Browser use

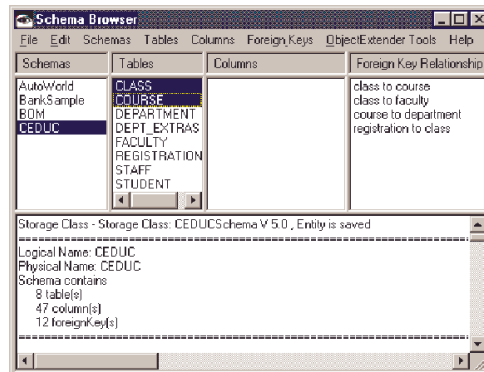
Describing a schema involves the usual tasks of defining any database schema, namely, creating tables, columns, primary keys, foreign keys and so on.

The database tables can be defined from scratch or they can be read, that is, imported, from an existing database.

Once you have described the schema, you can export the schema to the database, or you can generate DDL or scripts to invoke later to create the schema in the database.

Browser description

The Schema Browser is described below.



The Schema Browser presents several views:

- **Schemas.** Displays the names of your schemas .
- **Tables.** Displays the table names for a selected schema.
- **Columns.** Displays the column names for a selected table.
- **Foreign Key Relationships.** Displays the foreign key relationships for each table.
- **Information.** This view is not labeled as such, but it is a read-only view that provides descriptive information in a given context. For example, if you select only the name of a given schema, it will provide certain statistics about it.

Each view is used to progressively define your database schema. This involves defining database tables with their columns and key definitions.

Browser menu-bar choices

Most of the menu-bar choices for the Schema Browser are described below.

- **Schemas.** From this menu option, you can create, edit, and delete schemas. You can also import existing schemas from legacy databases. Exporting schemas to database is done from these menu options as well. The following choices are available.

New Schema

Start with this option when you are creating your own schema from scratch. You are prompted for a name. When you enter physical names for your tables, columns, and foreign keys, you must take into account any limitations of the underlying database. For example, the DB2 database currently limits table and column names to 18 characters. If you let ObjectExtender derive the physical names from the logical names, ObjectExtender makes useful truncations and solves duplicates by adding numbers. The maximum name length value can be set by executing the following Smalltalk code:

```
VapDatabasePhysicalRules maximumLength: 18.
```

Further customization of building physical names can be achieved through subclassing the *VapDatabasePhysicalRules* class.

Delete Schema

This option deletes the schema. If you have saved it previously then it is only deleted from the browser. You can reload it again if you change your mind later and want it back.

Import / Export Schema

This option has several choices and granularities for importing and exporting schemas, tables, and keys to and from the database, as well as providing the capability to drop tables and keys from the database. The choices are as follows:

Import Schema from Database

If you are working with legacy databases, this option will import the schema for you. You will be prompted to provide a name for the schema. You will be then prompted to provide the database connection information: **Connection Type**, **Data Source**, **Userid**, and **Password**. The user is then prompted with a dialog to select the desired tables to import. The user must select the appropriate qualifier and then press the Build Table List button. A choice list of available tables is then produced. The user should select the desired choices and then select OK

Alter Schema From Database

If you have made changes to an existing schema that you have previously exported, you can make minor changes with this option

Export Entire Schema to Database

If you are defining your schema from scratch, this option will export the schema to the database. You will be prompted to provide a name for the schema. Export uses the physical name. You will then be prompted to provide the database connection information: **Connection Type**, **Data Source**, **Userid**, and **Password**.

Export Schema Tables to Database

If you are adding new tables to existing databases, this option will export the tables to the database for you.

Export Schema Keys to Database

If you are adding new keys to the database, this option will export them for you.

Drop Schema Tables from Database

If you are eliminating tables from a database, this option will drop them for you.

Drop Schema Keys from Database

If you are eliminating keys a database, this option will drop them from for you.

Save Schema

Saves your schema definitions to an application and storage class you supply.

Load Available Schemas

Loads all available schemas defined by storage classes in your image.

Revert Selected Schemas

Reverts to the previously saved version of your schema.

Generate DDL Script for Schema Creation

Generates Data Definition Language scripts for executing in a database environment to generate your schema. The scripts are sent to the informational view in the browser. You need to highlight and execute them when you are ready to do so.

Generate Script for Schema Creation

Generates scripts for executing in your development environment to generate the schema in the database. The scripts are sent to the information view in the browser. You need to highlight and execute them when you are ready to create the schema in the database.

Generate Methods for Schema Creation

Generates the scripts and saves them to a class you have already created.

Generate Model From Schema

A model is generated directly from the selected schema.

- **Tables**

For creating, editing, deleting, and exporting database tables, the following choices are available:

Export Tables

This option enables you to the following selective tasks on tables and keys. An export uses the physical name of each element, including tables, columns and keys.

Export Selected Tables to Database

Choose specific tables to export.

Export Selected Tables with Keys to Database

Choose specific keys to export.

Drop Selected Tables to Database

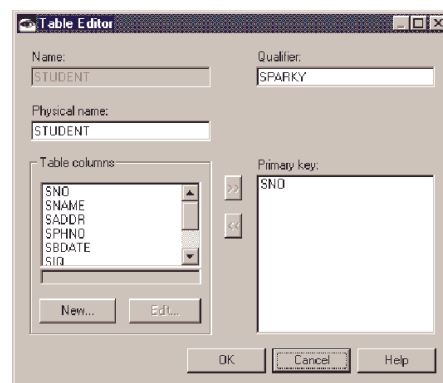
Choose specific tables to drop from the database.

Drop Selected Keys to Database

Choose specific keys to drop from the database.

New Table

This choice launches a Table Editor. Fill in the appropriate fields to complete the table definition.



The **Table Editor** prompts your for the following information:

- **Name.** This is a convenient logical name. Since most databases have limitations on their physical names, this logical name is a convenient way to provide a more descriptive name for the table.
- **Physical Name.** This is the physical name for the table as the database will know it. Some databases have limitations on the name length.
- **Qualifier.** Provide a qualifying name for the table. This is a unique identifier for the table used in conjunction with the table name to identify the table.
- **Table columns.** This is where you define the columns for the table. To define a new column, click **New**.
- **Primary key.** Define the primary key for the table by selecting a defined column name or names and clicking on the arrows.

You can also **Edit**, **Copy**, **Delete**, and **Rename** tables.

Generate DDL

Generates the Data Definition Language for the table for later uses.

- **Columns.** The following menu choices are available.

Columns

The Column Editor prompts you for the following:

- **Name.** This is a logical name relative to your object model. Since most databases have limitations on their physical names, this logical name is a convenient way to provide a more descriptive name for the column.
- **Physical Name.** This is the physical name for the column as the database will know it. Some databases have limitations on the name length.
- **Type details.** This information contains the atomic information for the column you are defining, that is, what type it will be, what kind of converter it will need, and if it can be null.
 - **Type.** From the database perspective, these are the types allowable that the database understands.
 - **Converter.** From the framework perspective, this is the converter that will execute to translate the database representation of the data into the object model representation.

Allow nulls. Check or uncheck if nulls are acceptable for the column.

Sort Columns

This is one choice of a pair of toggle switches. When selected, it is then disabled to indicate it is active.

Do Not Sort Columns

This is one choice of a pair of toggle switches. When selected, it is then disabled to indicate it is active.

Generate DDL

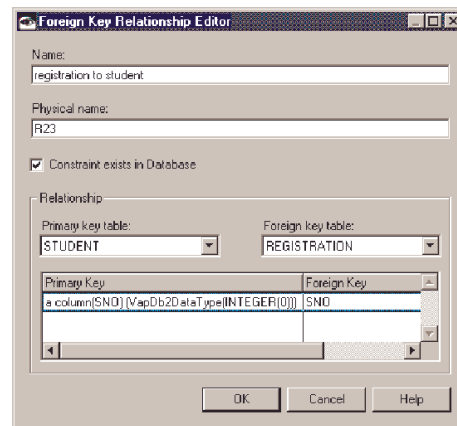
This option generated Data Definition Language for adding more columns to an existing table. The DDL is sent to the information view for highlighting and executing when ready.

- **Foreign_Keys**

For creating, editing and deleting foreign key relationships. The following choices are available:

Foreign Key Relationships

This option launches the Foreign Key Relationship Editor.



To define a foreign key, supply the following information:

- **Name.** This is a logical name. It does not store in the database.
- **Physical Name.** The physical name stores in the database and often has a length limit set by the database.
- **Constraint exists in database.**
- **Primary key table.** The primary key table is the table pointed to from the foreign key table in the foreign key relationship.
- **Foreign key table.** The foreign key table points to the primary key table by holding a foreign key whose value is equal to the primary key of the primary key table. The foreign key table is dependent on the primary key table to exist in the relationship.

Generate DDL

Generates the Data Definition Language for creating the foreign keys in the database. The DDL is sent to the information view for highlighting and executing when ready.

- **ObjectExtender Tools.** Other ObjectExtender tools and browsers can be launched from this menu.

Rename Element

The Rename Element option ensures that all pointers to the element are updated appropriately.

The Map Browser

The Map Browser is used to logically connect the object model description with the data store schema description. This enables the framework to generate the necessary service code for such things as SQL queries and other supporting code needed for persisting the model objects. The Map Browser with tools is used to create mappings between class attributes and table columns. Class associations must be mapped to table foreign keys.

- Browser use
- Browser description
- Browser menu-bar choices

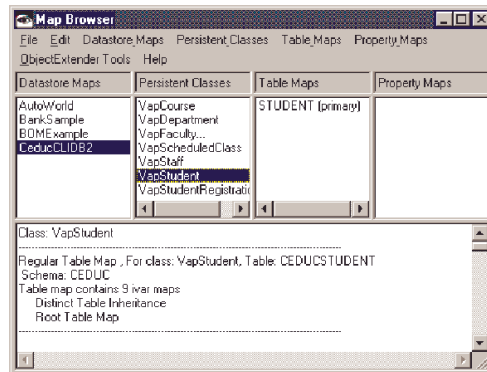
Browser use

Mapping an object model to a data store is done by creating table maps and property maps. A table map is required for the classes in your model that you want to persist in the data store. The property maps are defined for each table map by mapping object attributes to database columns.

In the Property Maps, two types of properties are shown. Maps prefixed with an **a** represent attribute property maps between attributes in the object model and columns. Maps prefixed with an **r** represent associations (**r** stands for relationship) between interclass associations and foreign key relationships in the database schema.

Browser description

The Map Browser is described below.



The Map Browser presents several views:

- **Datastore Maps.** Displays the names of your data store maps.
- **Persistent Classes.** Displays the names of your model classes which you will map to database tables.
- **Table Maps.** Displays the names of the table maps. These are the maps between your persistent classes and database tables.
- **Property Maps.** Displays the name of the property maps that are defined for each table map. The property maps included are the class attribute to database column mappings as well as class relationship to table relationship (connection) mappings.
- **Information.** This view is not labeled as such, but it is a read-only view that provides descriptive information in a given context. For example, if you select a map it will provide certain statistics about the map.

In the Map Browser, you describe the mapping of each persistent class. A map is required for each persistent model class.

For each attribute, you specify the columns which should be read to populate that instance variable.

For each relationship, you specify the foreign key relationship which is its persistent representation.

The first step in defining the mapping for a class is to define a table map to contain the column and relationship maps. A class must have at least one table map specifying the table to which the class will be mapped. Table maps contain attribute and relationship maps which refer to columns and keys within the mapped table.

Mapping rules allow fields that have field lengths to be mapped together as well as fields of different data types.

Browser menu-bar choices

Most of the menu-bar choices for the Map Browser are described below.

- **Datastore_Maps.**

The following choices are available:

- New Datastore Map**

- This launches the New Data Store Map dialog. Give your map a name, and select the model and schema to map.

- Delete Datastore Map**

- Deletes the selected map from the browser.

- Save Datastore Map**

- Saves the map to an application and storage class that you supply.

- Load Available Maps**

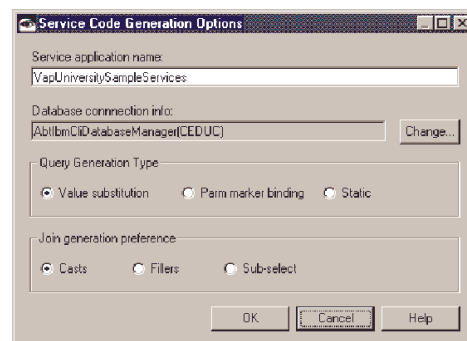
- This loads all available maps in your image.

- Revert Selected Map**

- Loads the previously saved version of the map selected. Use this option if you have made changes but want to discard them and go back to the original.

- Generation Options**

- Set default options for service code-generation.



- **Persistent_Classes.**

Enable pessimistic locking

This is a toggle setting that enables pessimistic locking for the selected model classes you are mapping to the data store.

Disable pessimistic locking

This is a toggle setting that disables pessimistic locking for the selected model classes you are mapping to the data store.

- **Table_Maps.**

New Table Map

Several kinds of maps are supported from this option as follows.

Table Map with No Inheritance

This maps a single class to a single table.

Add Single Table Inheritance Table Map

Known as typed partitioning, this maps a class hierarchy to a single database table.

Add Root/Leaf Inheritance Table Map

Known as vertical partitioning, this maps a class hierarchy to a hierarchy of database tables.

Add Secondary Table Map

Secondary maps are used when a single persistent class is mapped to more than one database table. For example, a persistent class *Department* may have *name*, *courses*, and *staff* mapped to the primary table (DEPT) and have other attributes such as *phoneNumber* mapped to a secondary table(DEPT_EXTRAS).

Edit Table Map

Delete Table Map

Edit Property Maps

Launches the Property Map Editor for modifications to property maps.

The Property Map Editor provides the means to map persistent class attributes and relationships to database columns and foreign key relationships. The editor contains two pages (tabbed dialog boxes): one for mapping attributes and one for mapping relationships.

The attributes page allows the user to choose between a *simple* mapping of an attribute (one attribute to one column) and a *complex* mapping (one attribute to one or more columns).

If you choose the *simple* map type option, you are prompted with a choice of columns and must pick one. If you choose a *complex* mapping then the Complex Attribute Editor is used to map the attribute. With the Complex Attribute Editor, you must complete two tasks:

1. Choose a *Composer* class.
2. Map each attribute of the complex attribute class to a database column.

Example: Suppose a model class has an attribute **address** of type *Address* and you wish to map the **number**, **street**, and **zip** attributes of the *Address* to three separate columns in the database. You would select the *complex* map type option from the Property Map Editor, then launch the Complex Attribute Editor. The editor requires you to choose a composer

class (*AddressComposer*). When the composer has been selected, provide the associated column for each of the *Address* attributes (**number**, **street**, **zip**).

The relationships page allows you to map class relationships to database foreign key relationships. This is a straightforward mapping along the lines of *simple* attribute mapping. You select a foreign key relationship for each class relationship.

- **Property_Maps.**

- Show inherited properties**

- Gives visibility to inherited properties when mapping tables with inheritance.

- Do not show inherited properties**

- Turns off visibility to inherited properties for inheritance table maps.

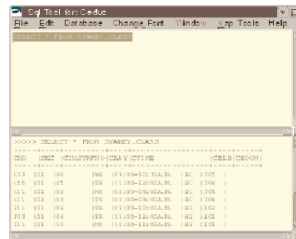
- Be part of optimistic predicate**

- Includes this attribute in a searched update.

The SQL Query Tool

The SQL Query Tool can be used to execute SQL code against the connection used by a generated *DataStore*.

The SQL Query Tool is described below.



The SQL Query Tool presents two views:

- Text entry view. In this view you can enter and execute SQL commands.
- Results view. This is a read-only view which displays the results of the SQL commands that you execute in the text entry view.

When you open the SQL Query Tool, you are prompted to supply the name of a data store and then a connection to the database is made. The name of the data store will be displayed on the window title.

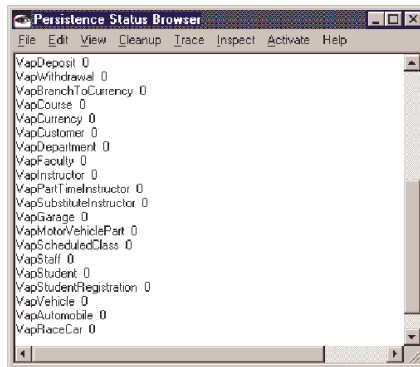
In the illustration of the tool, you can see that a database connection has been made with a data store named **CEDUC**. In addition, it is evident that an SQL query has been entered and the results displayed in the information view.

SQL commands are entered in the text view.

To execute the command you must highlight the command, and then choose the **Execute SQL** menu item from the text view's pop-up menu.

The Status Tool

The Status Tool can be used to monitor the transactions, views, and caches in the systems, and to reset the state of various components. It is intended to alert you to trouble-spots in your design such as performance problems and so on.



The Status Tool presents a single view where results of various system interrogations are supplied. For example, you can interrogate the state of your database connection, or analyze statistics on persistent objects, or home collections. The menu items are described below.

View Use this menu item to view **Home Collection Cache Statistics**, **Persistent Object Statistics**, **Data Store Statistics**, and **Transaction Statistics**.

Cleanup

This menu item allows you to perform cleanup functions including **Clear Home Collection Cache**, **Reset Data Store**, and **Release All Transactions**.

Trace This menu item allows you to perform tracing functions including **Basic Trace** and **Detailed Trace**.

Inspect

This menu item allows you to perform inspections including shared transaction, current transaction, transaction, service object, DO cache, relationship cache, and data store inspections.

Activate

This menu item allows you to **Activate Data Store**.

In the above illustration, the information shown in the results view was given when **Persistent Object Statistics** was selected from the **View** menu.

Chapter 5. Tasks

Tasks and samples overview

This section uses samples to help you complete each task. The following sample models are shipped with the ObjectExtender feature:

- AutoWorld
- Bank
- University

These models were constructed using the ObjectExtender tool set.

The samples require an installed database supported by this feature, such as DB2.

These models can be studied in-depth to learn how business objects and their relationships are defined, how specific objects are mapped to their data store, what their transaction isolation policies are, and so on.

If you prefer to do an example that gradually introduces you to the ObjectExtender framework, follow the one in “Your first ObjectExtender application”. This example leads you through a simple application using some of the business objects described in the University model.

Your first ObjectExtender application

Creating an ObjectExtender application can be divided into building:

1. The model
2. The persistency support for the model
3. The views that work with the model.

Each of these parts can be worked on by different people at different times and without impacting the other parts. This is one of the benefits of object-oriented programming whereby an application can be constructed in layers.

The first example is to create a simple model entity with some attributes, persist it, and then build some views to allow the entity to be listed, edited and deleted.

The next step is to then explore different ways of persisting the model as well as other ways of building views over the model.

The model that is used as an example is one of a university. This will include a number of entities including departments, faculties, students, course. To begin with the department will be defined, then persisted using DB2 as the data store, and then some VisualAge views built to allow departments to be listed, created, edited and deleted.

Creating a model

The first step is to create a model.

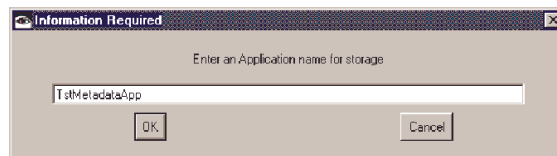
1. Open the Model Browser from the **ObjectExtender Tools** menu on the Transcript.
2. Select **New Model** from the **Models** menu.
3. Enter the name for the model: TstUniversity

Specifying storage details for the model

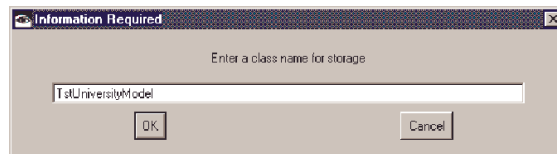
Each model has several storage entities associated with it that relate to how the model is stored in the Envy manager. These entities are the metadata application that is used to store the rules holding the information about how the model is defined.

To hold the metadata for the example model, an application named *TstMetadataApp* will be created. This should have a prerequisite of *VapMetadataApp*. The application is created when you save the model as follows:

1. Select the model, **TstUniversity**, in the **Models** view.
2. Select **Save Model** from the **Models** menu.
3. Enter the application name: *TstMetadataApp*.



Next, you are prompted to enter a class name. This class will hold all of the information about the **TstUniversity** model within the *TstMetadataApp*.



4. Enter the class name: *TstUniversityModel*

Because *TstMetadataApp* and *TstUniversityModel* are defined as classes in the Envy manager, they are able to benefit from all of the source management features that Envy provides, that is, they can be versioned to represent a baseline, exported from one manager to another, and included in configuration maps to allow developers to alternate between different consistent states of a model definition together with the Smalltalk classes that implement the model.

Once we have defined the storage entity for **TstUniversity**, we can save the model by selecting **Save Model** from the **Models** menu. This will create an edition of the class *TstUniversityModel* in the application *TstMetadataApp*.

Notes:

1. **Loading a model.** If another developer loads the application, *TstMetadataApp*, into their image they will not see the **TstUniversity** model in their Model Browser by default. The model must be loaded by selecting **Load Available Models** from the **Models** menu.
2. **Saving a model.** Saving a model will take the information changed in the browser and store it in an edition of the class **TstUniversityModel**.
3. **Reverting a model.** **Revert Selected Model** (on the **Models** menu) will revert the model to the last saved definition of the model in the *TstUniversityModel* class.

Creating a class in the model

The first entity that we will create in the **TstUniversity** model is the department model class. This is a relatively simple class with four attributes.

1. Select **TstUniversity** in the **Models** view.
2. Select **New Class** from the **Classes** menu.
3. Enter a class name: **TstDepartment**.

Add the four attributes as follows:

Attribute name	Attribute type	Value required
department	String	Yes
building	String	No
phone	String	No
room	String	No

For each attribute, do the following:

1. Click **New** in the **Attributes** tabbed dialog box. This launches the Attribute Editor.
2. Type the attribute name: **department**, for example.
3. Select **String** from the **Type** menu.
4. Select **OK**.

For an object to exist in ObjectExtender it must have a unique identifier. This is also known as an object identifier (OID). We will use the attribute, **department**, as the OID for the *Department* class.

To define an object identifier, do the following:

1. Select the attribute, **department**, in the **Attributes** view.
2. Click **>>**.

This displays the name of the attribute in the **Object ID** list.

This concludes creating the class, *TstDepartment*.

Save the model.

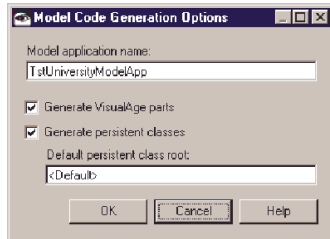
Creating the code for the model

The next step is to generate the Smalltalk classes that will support the *Department* class definition in our image. Before this can be done, we need to specify certain things regarding the kind of generation we wish to perform the application into which the generated Smalltalk classes will be stored.

Specifying the application into which the model is generated

Select the **TstUniversityModel** in the models view, then select **Model Code Generation Options** from the **Models** menu.

This launches the following dialog.



The name of the application used to store the generated Smalltalk classes that will make up the **TstUniversity** model should be *TstUniversityModelApp*. VisualAge will automatically create this application if it does not exist but any existing application can be used as long as it has a prerequisite of *VapPersistence*.

In addition to generating the Smalltalk code for the model classes, ObjectExtender is able to generate public interface features and methods that will allow the model classes to be used within a VisualAge composition editor. This should be selected (checked). Likewise we also will use the ObjectExtender feature that is able to generate persistent classes.

Generating the classes

The next step is to generate the Smalltalk classes that contain the code to support the execution of the **TstUniversity** model. Select **Generate** from the **Models** menu on the Model Browser.

The following classes will be generated:

TstDepartment

The business object class to support the behavior of the department.

TstDepartmentHome

The home collection class that has the protocol to create, list and find instances of *TstDepartment*

TstDepartmentKey

The concrete class that is used to represent the key object of a *Department*

Creating persistence support

Before proceeding further with the example we will create some persistence support. This will enable us to illustrate the features of the home collection class, the support for transactions within ObjectExtender, and to start building views. Good object-oriented applications should be separated into layers and the persistence layer has the knowledge to store our business objects (that is, make them persistent) as well as do lookups for us against the store.

The persistence support classes we will generate will enable us to store objects locally within our image. This helps to iterate over the model schema and the model classes and to build working views to get the application as far along as possible without having to think about the issues associated with mapping the business objects to a relational data store.

To generate the persistence support that will enable us to do local image persistence, do the following:

1. From the Model Browser, select **TstUniversity**, in the **Models** view.
2. Select **Generate Image Schema** from the **Models** menu.

The persistent support classes are generated into the application, *TstUniversityServicesApp*.

The following classes will be generated:

TstUniversityDataStore

The data store owns and manages a pool of data store connections (sessions) and registers a home collection for each data store connection.

TstUniversityTstDepartmentDataObject

The data object class contains the data for a business object in the form in which it was retrieved from the persistent store.

The data objects that support the business objects will be stored in the image, and saved when the image is saved. To clear out these objects, and start from an empty set of objects, evaluate the following code:

```
ImageServiceObject reset.
```

TstUniversityTstDepartmentServiceObject

The service object implements the Create/Read/Update/Delete (CRUD) protocol and other navigation methods required when mapping data objects to and from the persistent store.

Before working with objects in the model, the data store must be activated. To continue, evaluate the following code:

```
TstUniversityDataStore singleton activate.
```

Because we generated the schema for local image persistence, the superclass of *TstUniversityDataStore* is *LocalImageDataStore* which will use the image as the mechanism for persistence.

The department home collection class

The home collection class is a very important class that collaborates with the business object to provide a number of support services. It is a singleton class which means there is only ever one instance in the image at any given time.

The primary tasks of the home collection are to provide support protocol for creating, finding and accessing instances of the model class.

```
TstDepartmentHome singleton allInstances
```

This will return a collection of all *TstDepartment* objects. Note that this is different than the class method, *allInstances* which will actually query the image for all known instances of an object. The home collection object answers a collection of all instances that were created through the public creation protocol on the home collection itself. To illustrate this, evaluate the following code:

```
TstDepartment new.  
TstDepartmentHome singleton allInstances.
```

The department that was created in the first statement does not show up in the list of *allInstances* that the home collection returns. To create bona fide persistent instance of the *TstDepartment* class the create method should be used on the home collection.

Before working with our department objects we need to understand the transaction model. At any one point in time there is always a current transaction inside of which all of the work on business objects is done. A special transaction exists which is the shared transaction. The shared transaction represents the persisted view of the business model and is read only. To begin changing persisted objects a new transaction must be started. Each transaction has a parent which is the transaction into which it will commit its changes. A top-level transaction is one which has a parent of the shared transaction, that is, when it is committed its changes will be made persistent. Top-level transactions can be created with the following method:

```
Transaction begin
```

To create objects a transaction must be started. The transaction allows work to be undone and committed as a unit of work to the persistent store. To create data for our sample we need to evaluate the following code in the System Transcript window:

```
Transaction begin.  
(TstDepartmentHome singleton create)  
  department: 'Math';  
  room: 'A2'.  
Transaction current commit.
```

Now inspect the same code two more times to create English and History departments. Provide a room number for each department as well.

To see the committed department the home collection can be queried for *allInstances*.

```
TstDepartmentHome singleton allInstances.
```

Home collection also have "lookup" protocol to let us search for objects. Method to help locate an object by the key are generated automatically, that is, *TstDepartmentHome* has a method, *findByDepartment: aDepartmentString*.

```
| department |  
Transaction begin.  
department := TstDepartmentHome singleton findByDepartment: 'Math'.  
department room: 'A3'.  
Transaction current commit.
```

To delete an object it can be sent the unary method, *markRemoved*. This does not actually delete the object but rather marks it to be deleted in the current transaction and it is then deleted when the transaction is committed.

```
| department |  
Transaction begin.
```

```
department := TstDepartmentHome singleton findByDepartment: 'Math'.
department markRemoved.
Transaction current commit.
```

Creating a view

The next step is to create some views that will list departments, as well as create, delete, and edit departments.

In the Organizer, create a new application called *TstUniversityViews*. We will use this application to put the view classes for our university model. The first view we create will list departments and provide the capability to edit their details.

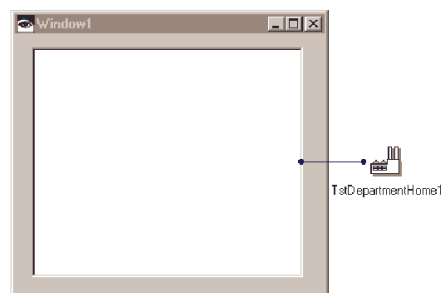
Listing departments

Create a new (visual) part for the *TstUniversityViews* application called *TstDepartmentView*.

Open the Composition Editor on *TstDepartmentView*. To display the departments, you will need to add a List part and a home collection part as follows:

1. Select the Lists category from the left column of the parts palette, and then select the List part from the right column of the parts palette and drop it in the Window part on the free-form surface.
2. Select **Add Part** from the **Options** menu, and type the class name, *TstDepartmentHome*.
3. Connect the *TstDepartmentHome* attribute *allInstances* to the *items* attribute of the List part.

This connection specifies that all department instances will display in the list.



Editing departments

To make the list editable is the next goal. To edit the details of a department, you will select the department in the list and modify its data in text fields.

Do the following:

1. Add a Variable part. You can use Object Extender visual parts on the parts palette or select **Add Part** from the **Options** menu, and type the class name, *TstDepartment*. Select **Variable**, under Part Type. Select **OK**.
2. Connect the *self* attribute of *TstDepartment* to the *selectedItem* attribute of the List part.

The variable will contain the object that represents the selected department as individual departments from the list are selected. By connecting the attributes of the department to some edit controls we can modify the details for a selected department.

If we were to do these steps an exception would be raised by the shared transaction stating that we attempted to write into a read-only version. This is because persisted objects cannot be modified unless a transaction has been started.

ObjectExtender has some visual parts to help you work with transactions. In the Composition Editor, the parts palette has a category named `! department`. Transaction begin. department := TstDepartmentHome singleton findByDepartment: 'Math'. department markRemoved. Transaction current commit. . The first ObjectExtender part to illustrate is the **TopLevelTransaction** part. This part will always begin a new top-level transaction when the view on which it is used is opened.

Do the following:

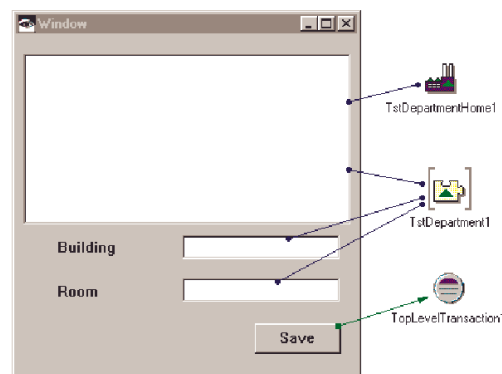
1. Add a TopLevelTransaction part on the free-form surface of the Composition Editor.
2. Add a Push Button part to the Window part.
3. Label the push button: **Save**.
4. Connect the push button's *clicked* event to the *commit* action of the TopLevelTransaction part.
5. Add two text boxes from the Data Entry controls and add corresponding labels for Building and Room. You can also use the Quick Form menu item and select the building and room attributes.
6. Test the view by opening it and selecting the Math department.
7. Change its building and room and then close the view.
8. Open the view again.

Note that the changes were lost. This is because the transaction was not committed.

9. Make some changes again and click **Save**.

This time the changes are committed to the shared transaction and made persistent.

Open it again and note that the changes were lost. This is because the transaction was not committed. Make some changes again and click **Save**. This time the changes are committed to the shared transaction and made persistent.



Creating departments

To create departments, we will add another button with the label, **New**. This will create a new department that is placed into the variable part. Do the following:

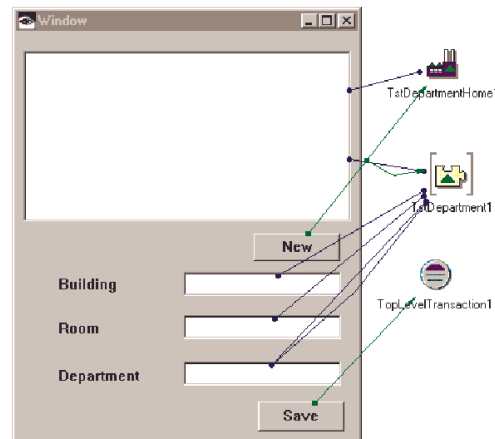
1. Add a Push Button part to the Window part.
2. Label the push button: **New**.
3. Connect the push button's *clicked* event to the *create* action of the TstDepartmentHome1 part.
4. Connect the *normal result* of the action, the newly created department, to the *self* attribute of the TstDepartment1 .

To create new departments, you will need to add a Text part, and a label, **Department**, for the *department* attribute. After you have added the Text part and label for department, connect the Text part's *object* attribute to the *department* attribute of the TstDepartment1 variable.

Test the view as follows:

1. Click the **Test** button.
2. Click **New** on the TstDepartmentView.
3. Type the name of a new department: Philosophy.
4. Type a room number for the department: 101.
5. Click **Save**.
6. Close the view.
7. Open the view again to verify that the new department has been added.

As before, if we create the department (click **New**) without saving it (click **Save**), the department will not be added because the transaction would not have been committed.



The next step is to get the list of departments to refresh each time we add or change a department. Also, we should ensure that the department itself cannot be changed when we are editing a department. This is because the department is the attribute we defined as the object identifier (OID) in the Map Browser, and it is the effective key of the department business object. Changing the key of a business object is not permitted once an object has been persisted because it means we will no longer be able to retrieve it from the data store.

To ensure that the department's key cannot be changed for an existing department do the following:

- Connect the *isNotPersistent* attribute of the TstDepartment1 part to the *enabled* attribute of the *department* Text part.

In this way, the Text part is only enabled when creating a department. When an item is selected in the list, it means we are editing an existing persisted department; its object identifier therefore cannot be modified.

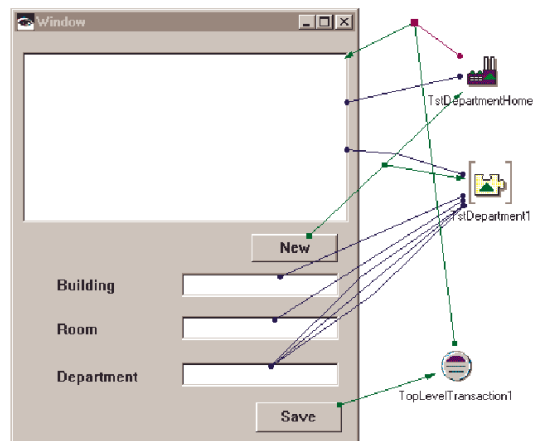
To get the List part to refresh, we can do the following:

- Connect the *committed* event of the TopLevelTransaction1 part to the *items* action of the List part.

The connection wire should be dotted indicating that we need to supply a parameter. The parameter we need to supply is the *allInstances* attribute of the TstDepartmentHome1 part. Connect *value* to *allInstances*.

Whenever the transaction is committed, the items of the List part are therefore refreshed with the *allInstances* of the *TstDepartment* class.

- In the illustration, the event-to-action connection from the TopLevelTransaction part to the List part is dragged out to the right to help make the view more readable.



Deleting departments

Thus far, we can create departments, and edit existing ones. Deleting departments is the next task we want to build into our application. Do the following:

1. Drop a Push Button part on the Window part and label it: **Delete**
2. Connect the *clicked* event of the **Delete** button to the *remove* action of the TstDepartment1 variable.

All business objects have an action *remove* that will mark the object as removed. Pressing **Save** after **Delete** would achieve this but it is probably easier to commit the transaction immediately after you click **Delete**.

3. Connect the clicked event of the **Delete** button to the commit action of the TopLevelTransaction1 part.

The last step is to ensure that the **Delete** button is disabled until there is actually an object to delete. This can be done by connecting the *selectionIsValid* attribute of the List part to the *enabled* attribute of the **Delete** button.

4. Test the view. It is possible now to add, edit, and delete departments.

Do the following:

1. From the ObjectExtender parts palette category, drop a SharedTransaction part onto the free-form surface.
2. Drop a **variable** part for TopLevelTransaction onto the free-form surface. This will replace the TopLevelTransaction part.
3. Reconnect the TopLevelTransaction part connections to the TopLevelTransaction variable part, then delete the original TopLevelTransaction part.
4. Connect the *openedWidget* event of the Window part to the *beginChild* action of the SharedTransaction and the result of this to the *self* attribute of the TopLevelTransaction variable.

This ensures that when the view opens we have a top-level transaction to work with.

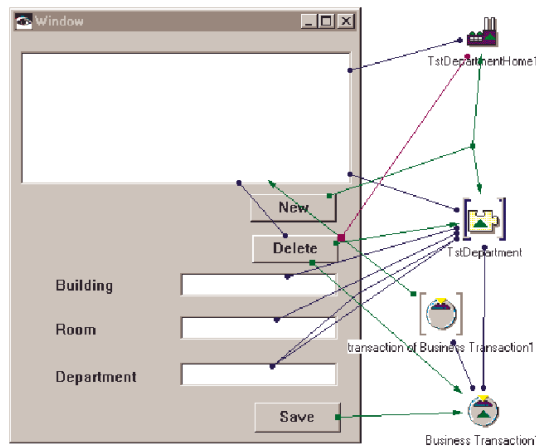
5. Connect the *committed* event of the TopLevelTransaction variable to the *beginChild* action of the SharedTransaction part.

This connection will generate a fresh top-level transaction each time one is saved. Connect the result of the connection to the self attribute of the TopLevelTransaction variable.

There are two connections from the *committed* event of the TopLevelTransaction variable. One of these is to the SharedTransaction part to regenerate a fresh transaction and the other is to the *items* attribute of the List part to ensure that is refreshed. It is important to ensure that the connection to the SharedTransaction part is the first of these two that happens.

1. Select the TopLevelTransaction part and from its pop-up menu, choose **Reorder Connection From**.
2. Drag the connections to make sure that the connections fire in the right order.

You can also use a Business Transaction in place of the Top-Level Transaction.



Note: When using ObjectExtender with Web Connection parts you must be particularly aware of transaction scope. When possible, the scope of the transaction should be limited to the equivalent of one page. If this design does not fit your application, then you will need to place a handle to the current transaction in session data. Each Web Connection part will then need to retrieve the current transaction from the session data. This is best accomplished by writing scripts.

Transactions in more depth

The example thus far works in a scenario with only one transaction. This is the top-level transaction that is created initially when the view opens and regenerated from the shared transaction each time it is committed. It is possible, however, to have more than one transaction in existence at the same time. At any one point in time only one transaction can be the current transaction. Whenever a transaction is created it becomes the current one and whenever a transaction is committed or rolled-back, the parent of the committed transaction will become the current transaction after the commit. To illustrate this consider the following code:

```
Transaction reset
```

This will reset all transactions back to an initial state. Only the shared transaction will exist.

```
Transaction current
```

The above therefore will return the shared transaction. This can also be done with the explicit message:

```
Transaction shared
```

When a new top-level transaction is started, it is the current transaction. For example,

```
Transaction begin.  
Transaction current.
```

The current transaction is now the top-level transaction.

```
Transaction current rollback.  
Transaction current.
```

The above code would roll back the top-level transaction, and the current transaction would now be the shared transaction (the parent of the top-level transaction).

Nested transactions

All transactions have a parent transaction. This is the transaction from which they were created with the *beginChild* method.

The shared transaction has no parent transaction. It represents a persisted view of the world. It could be thought as a proxy or gateway to the actual database which is shared with other users.

The top-level transaction can be thought of as a special transaction which is a child of the shared transaction. When it is committed, its changes are merged into the shared transaction where they become persisted and permanent.

It is not just the shared transaction which can create child transactions. Any transaction can create child transactions providing support for different levels of nested transactions. The following code creates a top-level transaction, and then creates a child of the top-level transaction. A department is created in the top-level child and committed through to the shared transaction.

```
| topLevel topLevelChild |
topLevel := Transaction begin.
topLevelChild := topLevel beginChild.
TstDepartmentHome singleton create department: 'Geography'.
topLevelChild commit.
```

When the top-level child is committed it merges its changes into its parent, in the above case, the parent is the top-level transaction. Inspecting *allInstances* from the home collection, you will see that the Geography department has not yet been persisted.

```
TstDepartmentHome singleton allInstances.
```

If the top-level transaction is committed, the changes are applied to the shared transaction, and the new department will become persisted.

```
| topLevel topLevelChild |
topLevel := Transaction begin.
topLevelChild := topLevel beginChild.
TstDepartmentHome singleton create department: 'Geography'.
topLevelChild commit.
topLevel commit.
```

Intuitively, if we did not commit the top-level child, but only committed the top-level transaction, we might expect that the Geography department would not become persisted. We have not explicitly committed the top-level child transaction (the one in which the department was created) and therefore it has not been applied to the top-level transaction. However, when a transaction is committed, all of its child transactions are also committed.

```
| topLevel topLevelChild |
topLevel := Transaction begin.
topLevelChild := topLevel beginChild.
```

```
TstDepartmentHome singleton create department: 'Geography'.
topLevel commit.
```

The result of this is that the Geography department is persisted and becomes visible in the shared transaction.

```
TstDepartmentHome singleton allInstances.
```

When a transaction is committed, it first commits all of its child transactions in the order they were created, and then commits itself. Thus, by committing the *topLevel* transaction, the Geography department is first pushed from the *topLevelChild* into the *topLevel* and from there to the shared transaction.

Switching between transactions

In scenarios where there is more than one transaction being used, it is sometimes necessary to switch back and forth between them. Any transaction can be made the current transaction with the *resume* message.

The following example code illustrates this.

```
| transaction1 transaction2 |
transaction1 := Transaction begin: 'firstTransaction'.
transaction2 := Transaction begin: 'secondTransaction'.
Transaction current inspect.
```

The inspection shows that *transaction2* is the current transaction.

```
| transaction1 transaction2 |
transaction1 := Transaction begin: 'firstTransaction'.
transaction2 := Transaction begin: 'secondTransaction'.
transaction1 resume.
Transaction current inspect.
```

In the above example, *transaction1* is the current transaction because it was explicitly resumed.

```
| transaction1 transaction2 |
transaction1 := Transaction begin: 'firstTransaction'.
transaction2 := Transaction begin: 'secondTransaction'.
transaction2 suspend.
Transaction current inspect.
```

If a transaction is asked to suspend itself, the shared transaction will become the current transaction. This occurs only if the transaction being asked to suspend itself is actually the current transaction at the time it is suspended.

```
| transaction1 transaction2 |  
transaction1 := Transaction begin: 'firstTransaction'.  
transaction2 := Transaction begin: 'secondTransaction'.  
transaction1 suspend.  
Transaction current inspect.
```

In the above example, *transaction2* is the current transaction after it is created, and *transaction1* therefore cannot be suspended; *transaction2* remains as the current transaction.

Two top-level transactions

The following scenario creates two top-level transactions and switches between them. The Math department is created in the first transaction, the French department in the second transaction. After the first transaction is committed, the inspection of all department instances shows only the Math department, then the second department is committed and both are visible when inspected.

```
| firstTransaction secondTransaction |  
firstTransaction := Transaction begin: 'firstTransaction'.  
TstDepartmentHome singleton create department: 'Math'.  
secondTransaction := Transaction begin: 'secondTransaction'.  
TstDepartmentHome singleton create department: 'French'.  
firstTransaction commit.  
Transaction current allInstances inspect.
```

At this point only the 'Math' department has been committed.

```
Transaction current.
```

The shared transaction is the current transaction at this point. It was the parent of the *firstTransaction*.

```
secondTransaction commit.  
TstDepartmentHome singleton allInstances inspect.
```

Now both the 'Math' and 'French' department have been committed.

Visual programming for more than one transaction

The *TstDepartmentView* built earlier uses only one transaction. This is the top-level transaction that it creates when it opens and regenerates each time it is committed. There is no guarantee however that this top-level transaction is the current transaction in a more complex scenario. Run the *TstDepartmentView* and create and edit some departments. Then switch to the System transcript (without closing the *TstDepartmentView*) and make the shared transaction the current transaction with the following code:

```
Transaction shared resume
```

Switch back to the open instance of *TstDepartmentView* and try to create or edit some department. You will get a walkback telling you that are trying to create or modify objects in the shared transaction. This is because we explicitly switched the shared transaction to be the current transaction. Access to both the *TstDepartmentHome* part and the *TstDepartment1* variable are done in the context of the current transaction and, because the shared transaction is read-only, the view will not work properly.

One way to program around this is to make the top-level transaction resume when the *TstDepartmentView* receives focus. This can be done with a connection from the *gettingFocus* event of the window to the *resume* action of the *TstDepartment1* variable.

The problem with relying on the *gettingFocus* event is that it assumes that all work done on the *TstDepartmentView* is while it has focus. In more advanced scenarios it is possible that the view will ask questions of its objects without focus being received, for example to repaint or refresh an area of the screen following an expose event. It is also possible that more than one transaction can be going on together on the same view. The solution to this is to have a mechanism of specifying at design time an association between a part and a transaction and having the guarantee that all access to the part is done within the context of that transaction. This is provided for with the part **TransactedVariable**. This can be found on the palette, under the ObjectExtender category.

Using the TransactedVariable part

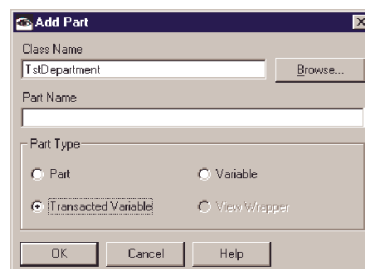
The first example of using the transacted variable part will be to make the *TstDepartmentView* transaction safe, that is, it will work in the correct transaction regardless of whatever else is going on around it.

Making TstDepartmentView transaction safe

We will change the *TstDepartmentView* created earlier such that it shows details of the department in both the top-level transaction and also the shared transaction.

The part **TransactedVariable** is a special type of variable part. It has all the behavior of a variable except that it has an additional attribute *transaction*. When a variable is given a transaction it will ensure that all actions, events and attributes performed on the variable are within that transaction. For the *TstDepartmentView* we need two transacted variables, one that has the department within the shared transaction, and one that has the department within the top-level transaction.

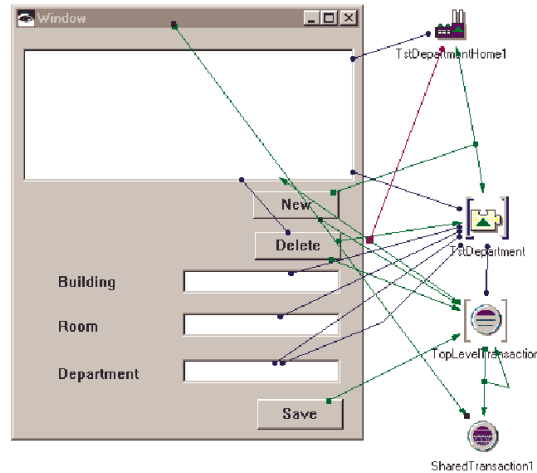
1. Reopen the Composition Editor on *TstDepartmentView*.
2. Select **Add Part** from the **Options** menu, and type *TstDepartment* for the class name.
3. Select **Transacted Variable**, under **Part Type**.



4. Move the connections you made for the *TstDepartment1* variable part to the transacted variable part, then delete the *TstDepartment1* variable part.

5. Connect the *self* attribute of the TopLevelTransaction2 variable to the *transaction* attribute of the transacted variable.

This will ensure that the variable TstDepartment1 always sets the attribute values of its contents, the transaction, in the top-level transaction.



Now, test the view. Switch to the Transcript and get the shared transaction to resume itself as described earlier. When you go back to the view and edit some attributes of the selected department, the earlier walkback, caused by attempting to update the shared transaction, no longer occurs. This is because of the connection from the TopLevelTransaction variable to the transaction attribute of the TstDepartment variable. This ensures that access to the variable's value is always done in the context of the specified transaction.

Go back to the System Transcript and look at the current transaction using the method

```
Transaction current.
```

Note that it will be the top-level transaction (if you took the step to resume the shared transaction earlier). The transacted variable part will ensure its contents are always accessed in the context of its specified transaction but it will not change the current transaction.

In addition to specifying the transaction in which the TstDepartment1 variable should operate, we should also specify the same for the department home collection. When the **New** is clicked, it performs the *create* action, thus the new department should be created in the top-level transaction.

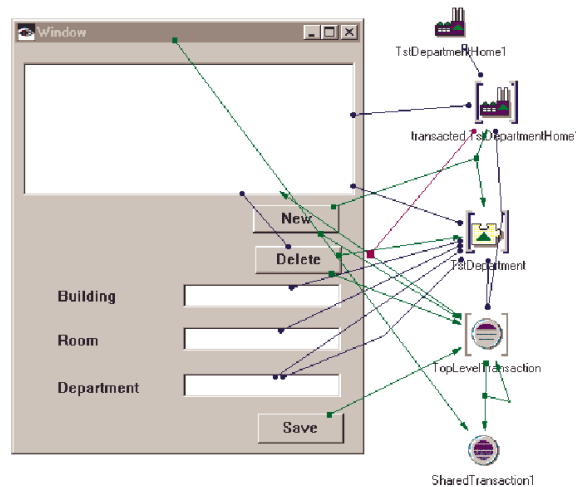
To create a transacted variable that contains the TstDepartmentHome1 part

1. Select the TstDepartmentHome1 part, and from its pop-up menu, select **Create Transacted Variable**, then drop the transacted variable on the free-form surface.

The transacted variable is then added for you and the connection from the *self* attribute of the TstDepartmentHome1 part to the *new* variable that will be named **transacted TstDepartmentHome1**.

2. Connect the *self* attribute of the TopLevelTransaction1 variable to the *transaction* attribute of this transacted variable and move all the connections from the TstDepartmentHome1 part to the newly created transacted TstDepartmentHome1 variable (except for the connection from the part to the variable itself).

As was the case earlier, make sure that the ordering of the *committed* events from the TopLevelTransaction are such that the connection to the SharedTransaction part is first.



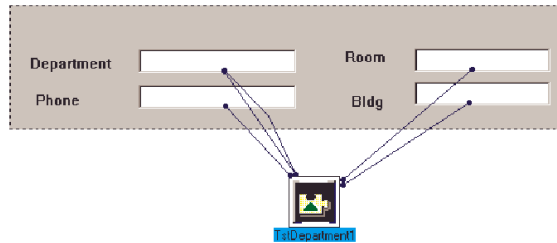
This view now ensures that the TstDepartment1 variable and the **transacted TstDepartment1** variable are always accessed within the context of the transaction within the TopLevelTransaction1 variable, regardless of whatever else is going on around the view in terms of which is the current transaction.

Viewing multiple transactions

To show the usage of the transacted variable part further, we will build a view that shows the attributes for a department in both the shared transaction and the top-level transaction. To do this, another transacted variable must be added that contains the selected department object. This variable must be given the shared transaction as its transaction.

To show the shared and top-level transactions, we will use multiple views. To show the multiple views, we will create a form with the edit controls and connections that we used formerly to show the details of the department. In addition, we will add a transacted variable to the form.

Furthermore, we will promote the following attributes of TstDepartment1: *self*, *transaction*, and *remove*. This will allow views that use this part to specify the department that it should work with as well as the transaction in which it should operate.

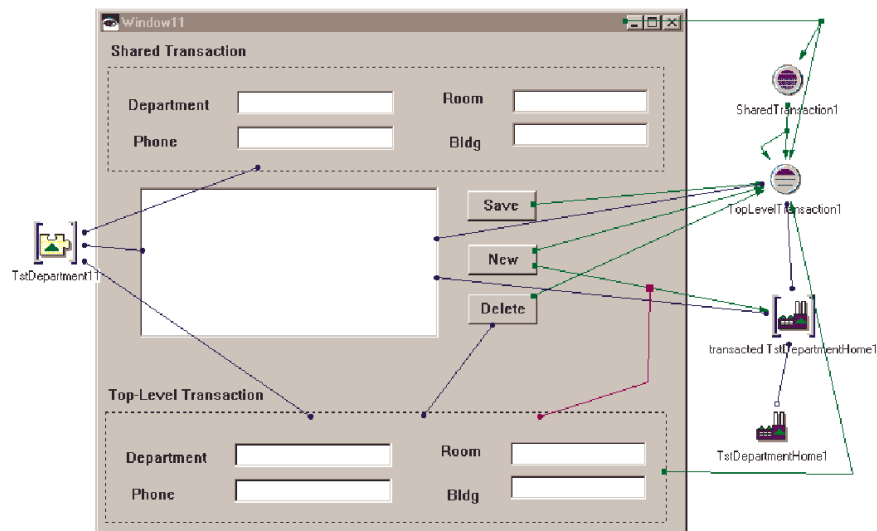


Create a new view, *TstDepartmentTwoTransactionsView*. Do the following:

- Add a List part.
- Add Push button parts for New, Delete, and Save operations as you did for *TstDepartmentView*.
- Add two *TstDepartmentEditForm* parts, one above, one below the List part.

The top *TstDepartmentEditForm* will be used to show the selected department in the shared transaction. When the form was built the attributes *self* of the *TstDepartment1* variable (representing the department which will be displayed) and the attribute *transaction* were promoted.

- Add a SharedTransaction part and connect its *self* attribute to the *tstDepartment1Transaction* attribute (the promoted transaction) of the *TstDepartmentEditForm*



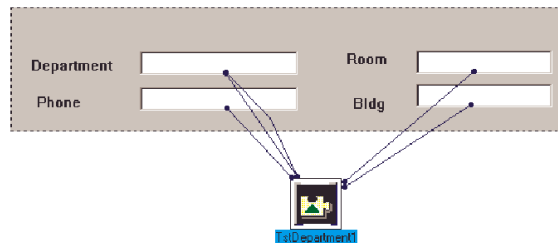
Transacted variables in editable container parts

The next example using transacted variable parts will be illustrated with editable container parts. We will build a view that shows the list of all departments in a Container Details part in which the values can be edited directly in the container cells. We will show the departments that can be modified in the top-level transaction, we will have a non-editable container that shows the shared transaction.

1. Create a new view called *TstDepartmentEditableContainersView* from the VisualAge organizer.
2. Add a Container Details part, and four Container Details columns.

3. Make these columns correspond to the attributes: *department*, *phone*, *room*, and *building*.
4. Open the settings of the Container Details part and set the **editable** property to be **false**.
5. Add the *TstDepartmentHome* part and create a transacted variable.
6. Connect the *allInstances* attribute from the transacted variable to the *items* of the Container Details part.

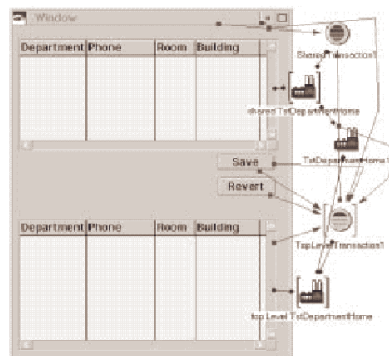
The next step is to ensure that the transacted variable will surface the collection of *TstDepartment* objects in the context of the shared transaction. By default, this will occur because if a transacted variable is not given an explicit transaction it will always use the shared transaction. However, we can also explicitly specify this by adding the shared transaction part and connecting its *self* attribute to the transaction attribute of the transacted variable containing the department home collection.



The next step is to add another Container Details part that will allow the departments to be listed.

1. Resize the Window part to make room for another Container Details part.
2. Create a new transacted variable from the *TstDepartmentHome* part and name this part **top-level department home**.
3. Set the editable property to **true** for the Container Details part.
4. Set the editable property to **false** for the department column.

Recall that department is the OID or key attribute for the department class and that key attributes cannot be modified.



Model to model relationships

The *TstUniversity* model so far has just included the *TstDepartment* class. This class has four attributes which are all of type *String*. The example will now be extended to cover relationships.

Note that some of the examples in this section may require code patches to actually run. Check the web site for any updates. The patches however are not required to build the examples.

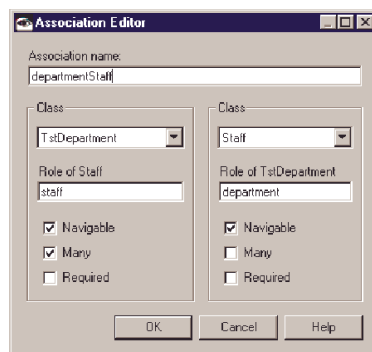
One-to-many relationship

We will create a model class called TstStaff and make a one to many relationship between TstDepartment and TstStaff. This way a department can have many staff and a staff member belongs to one department.

From the model browser create the TstStaff model. Give it the following attributes:

name	String
number	Integer
salary	ScaledDecimal
title	String

To add a relationship between the TstDepartment and TstStaff model classes select the menu bar option Associations and select the menu choice New Associations. Each relationship has a unique name, in this case it can be called departmentStaff. The classes at either end of the relationship are specified as well as the role they have with their counterpart. For example, aTstDepartment knows its Staff instances with the role staff and a Staff object knows its TstDepartment instances with the role department. The role name for the end of a many relationship is usually a plural word, for example, customers, departments, or staff. Our relationship is navigable from both ends in which case the Department will have a collection of Staff instances and each Staff instance can have a department. The 'Role of Staff' within the TstDepartment are set to be many such that a Department can have more than one staff instances. The 'Role Of TstDepartment' is not set to many which implies that it is a one relationship, i.e. a Staff instance can only have one TstDepartment across the relationship Neither end of the relationship is mandatory so the Required check boxes are not set.



Having added the relationship the model should be saved and regenerated.

It is important to get good names for the role names as these are used to generate methods. Generating the model will add the methods staff, addStaff: and removeStaff: to the TstDepartment class, and the methods department and department: to the **TstStaff** class.

Reminder: Whenever you generate or regenerate a model, you will need to generate or regenerate the services code as well.

Maintaining staff

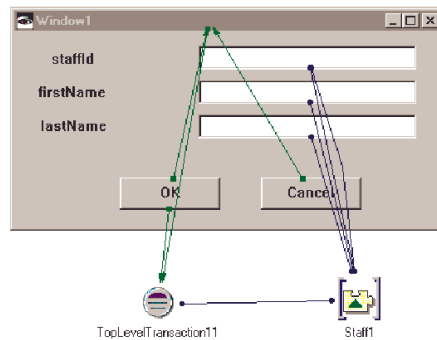
The first screen that we will build will allow staff to be maintained. This will be a list screen and a separate screen to add and edit the staff. The first screen to build is the edit screen. For this create a visual part named `TstStaffEditView`. Build the screen as described below.

Understanding the notation: Instructions in this section do not follow the format used thus far. Instead, the following devices are used:

1. **Annotated diagrams:** The user-interface is annotated with a series of numbers next to each part and connection.

The numbers indicate the following:

- Parts are numbered sequentially using whole numbers.
 - The numbers serve as indices in the reference tables.
 - For parts with multiple connections, the indices in the table also specify the order in which the connections must be made.
2. **Reference tables:** The tables specify what you need to know about the parts: their names, property settings, and connections.
 3. **Comments:** The comments provide important details, such as if you need to promote a part. Any further information that may be helpful toward a better understanding of the feature is also provided in this space.



Parts List		
Part Index	Part Class	Part Name
1	AbtLabelView	Label1
2	AbtShellView	Window
3	AbtLabelView	Label2
4	AbtTextView	Text1
5	AbtLabelView	Label3
6	AbtTextView	Text2
7	AbtPushButtonView	PushButton1
8	AbtTextView	Text2
9	AbtTransactedVariable	Staff1
10	AbtPushButtonView	PushButton2
11	AbtVariable	TopLevelTransaction1

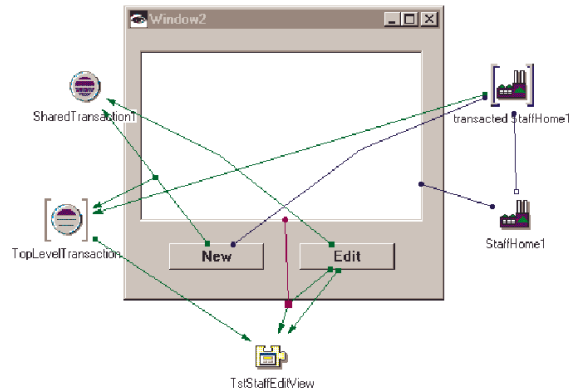
Property Settings

Part Index	Property	Value
1	Object	StaffId
3	Object	FirstName
7	Object	Cancel
8	Object	LastName
9	PartType	Staff
10	Object	OK
11	PartType	TopLevelTransaction

Connections			
Source Index	Source Feature	Target Index	Target Feature
9	staffId	1	object
9	firstName	3	object
9	lastName	8	object
9	isNotPersisted	3	enabled
11	self	9	transaction
7	clicked	2	closeWidget
10	clicked	3	commit
10	clicked	2	closeWidget
2	closedWidget	11	rollback

Comments: Promote the self feature of the TopLevelTransaction1 variable and also the self feature of the Staff1 transacted variable. When this view is launched it can be passed a Staff object and a top level transaction and it will allow the staffId, firstName and lastName of the Staff object to be shown and modified. The OK button commits the transaction and shuts the view down, and the cancel button shuts the view down. When the view has been closed the transaction is rolledback as it is no longer required. It is important that the two connections from the OK button are in the correct order, that is, the transaction is committed and then the view closed. If they were the other way around the view would close itself and rollback the transaction before it was committed and the changes would be lost.

The next view that should be build is the one that will allow the existing staff to be listed and new ones to be added and edited. Create a visual part called **TstStaffMaintenanceView** and build it as follows:



Parts List		
Part Index	Parts List	Part Name
1	AbtListView	List1
2	AbtShellView	Window
3	AbtPushButtonView	PushButton2
4	AbtPushButtonView	PushButton1
5	StaffHome (HomeCollection)	StaffHome1
6	AbtAppBldrViewWrapperEditPart	TstStaffEditView
7	AbtTransactedVariable	transacted StaffHome1 (see comments)
8	SharedTransaction	SharedTransaction1
9	AbtVariable	TopLevelTransaction1

The transacted variable (7) is created by clicking on the StaffHome to bring up its pop-up menu an selecting **Create Transacted Variable**.

Property Settings		
Part Index	Property	Value
3	Object	Edit
4	Object	New
5	PartType	StaffHome
6	PartType	TstStaffEditView
7	See comments section	
8	PartType	SharedTransaction
9	PartType	TopLevelTransaction

Comments: As mentioned in the comments above, this part is generated automatically.

Connections			
Source Index	Source Feature	Target Index	Target Feature

5	allInstances	1	items
5	self	7	self (see comments)
4	clicked	8	beginChild
4	clicked	6	openWidget
4	clicked	7	create
3	clicked	8	beginChild
3	clicked	6	openWidget
3	clicked	6	staff1
9	self	7	transaction

Comments: In the above connections:

1. Each clicked event connects normalResult to the TopLevelTransaction. In addition, the clicked event for the edit operation takes the value parameter from the selectedItem of the list.
2. The connection between home collection and the transacted variable part (self-self) is done automatically when you create the transacted variable.

Maintaining the department to staff relationship

We can now maintain Department and Staff separately. The next step is to allow the relationship to be maintained. Assume that we have a department called 'Geography' and a staff with id of 15. If we want to create a relationship between the two, use the code as follows:

```
| dept staff |
Transaction begin.
dept := TstDepartmentHome singleton findByDepartment: 'Geography'.
staff := StaffHome singleton findByStaffId: '15'.
staff dept: dept.
Transaction current commit.
```

All we did in the code was to call the department: set method on the staff object. Run the following code

```
( TstDepartmentHome singleton findByDepartment: 'Geography' ) staff
```

Note that the staff will appear in the department's staff. The inverse link of the relationship has been maintained. This is a very powerful feature of ObjectExtender whereby because the relationship was defined to the model browser with the knowledge of both ends of the relationship only one end of the relationship has to be publicly maintained and the other end will be modified automatically. This is also known as relationship handshaking.

To undo the relationship there are therefore two ways for it to be done, either by setting the department to nil on the staff object, or else using the removeStaff: method on the department. The example below uses the removeStaff: method:

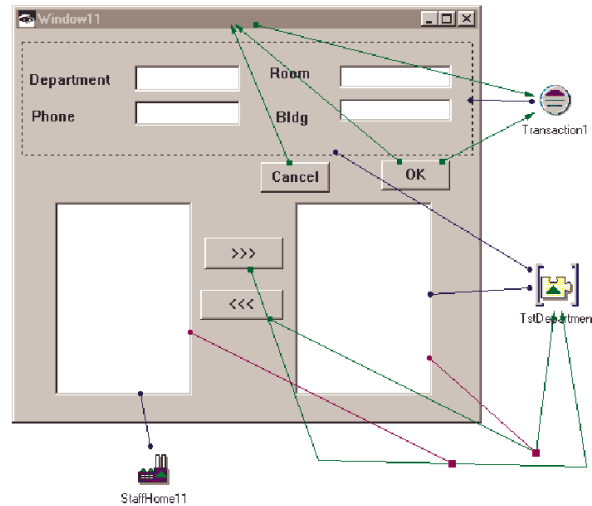
```
| dept staff |
Transaction begin.
dept := TstDepartmentHome singleton findByDepartment: 'Geography'.
```

```

staff := StaffHome singleton findByStaffId: '15'.
dept removeStaff: staff.
Transaction current commit.

```

For the user interface we will allow the user to work with a department and add and remove staff members. Modify the TstDepartmentEditView to add two list boxes at the bottom. The left hand list box will show a list of all Staff instances from the StaffHome. The right hand list box will show all staff from the staff relationship attribute of the TstDepartment object. Two buttons allow staff to be added and removed to the TstDepartment.



This concludes the relationship example.

Creating relationships

This section uses the Bank sample model, one of the several sample models that are shipped with ObjectExtender. The topics explain how to construct one-to-one, one-to-many, and many-to-many relationships.

Creating one-to-one (1-1) relationships

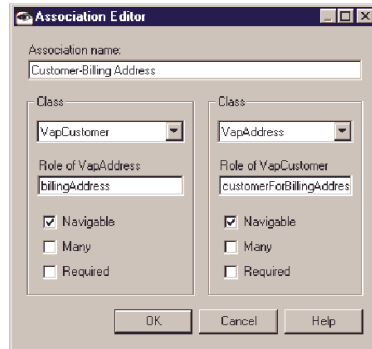
The Bank sample uses a one-to-one relationship for the *VapAddress* and the *VapCustomer* business objects. This topic explains how to create a one-to-one relationship using the ObjectExtender tools.

Do the following:

- Launch the Model Browser, and select **Bank** from the **Models** view.
The **Customer-BillingAddress** class association is an example of a one-to-one relationship.

It was created as follows:

1. Select **New Association** from the **Associations** menu.
This launches the Association Editor.



2. In the **Association name** field, type Customer-BillingAddress.
3. Define the two **Class** sections.

- a. In the left **Class** pane, select *VapCustomer*.
- b. In the right **Class** pane, select *VapAddress*.
- c. Type billingAddress for **Role of VapAddress**.
 - Select **Navigable**.

This means *VapAddress* can be obtained from the *VapCustomer* object.

Leave **Many** and **Required** unselected. This sets a cardinality of zero-to-one (0:1). When a cardinality of one-to-one (1:1) is desired, select **Required**.

- d. Type customerForBillingAddress for **Role of VapCustomer**.
- e. Select **Navigable**.

This means *VapCustomer* can be obtained from the *VapAddress* object.

Leave **Many** and **Required** unselected.

4. Select **OK**.

You are now done with the Model Browser. The one-to-one relationship is defined.

Next, defining this relationship using schema semantics is shown. Mapping the schema to the model will be the last step covered. These combined tasks provide the required information for creating persistence support for your business objects.

Do the following:

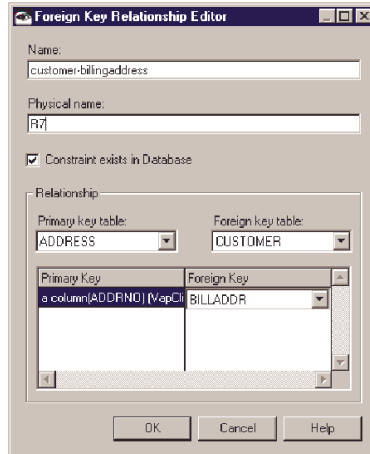
- Launch the Schema Browser, and select **Bank Sample** from the **Schemas** view, and select **CUSTOMER** from the **Tables** view.

The **customer-billingaddress** foreign key relationship is the one-to-one relationship that corresponds to the Customer-BillingAddress class association (relationship).

It was created as follows:

1. Select **Foreign Key Relationship** from the **Foreign_Keys** menu.

This launches the Foreign Key Relationship editor.



2. Type **customer-billingaddress** for the **Name**.
3. Type **R7** for the **Physical Name**.
4. If a foreign key constraint does not exist on the database, make sure **Constraint exists in database** is not selected.
5. Update the **Relationship** section:
 - a. Select **ADDRESS** for the **Primary key table**.
The **Primary key** column (read-only) will be updated with the primary key from the table.
 - b. Select **CUSTOMER** for the **Foreign key table**.
In the **Foreign key** column, select the foreign key in the **CUSTOMER** table which corresponds to the billing address, **BILLADDR**.
6. Select **OK**.

You are now done with the Schema Browser.

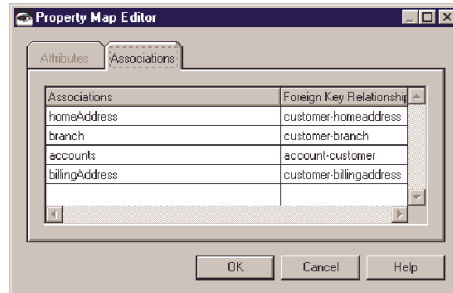
To complete the last step in defining the persistence layer for the one-to-one relationship.

- Launch the Map Browser. Select **Bank Sample** from the **Datastore Maps** view, and select **VapCustomer** from the **Persistent Classes** view, and then select **CUSTOMER** from the **Table Maps** view.

The **(r) billingAddress(customer-billingAddress)** property map represents the mapping from the **billingAddress** attribute from *VapCustomer* to the **customer-billingAddress** foreign key relationship.

It was created as follows:

1. Select **Bank Sample** from the **Datastore Maps** view, **VapCustomer** from the **Persistent Classes** view, and **CUSTOMER** from the **Table Maps** view.
2. Select **Edit Property Maps** from the **Table_Maps** menu.
This launches the Property Map Editor.
3. Click the **Associations** tab.



4. Change the [Not Mapped] value of the **billingAddress** association to **customer-billingaddress** under **Foreign Key Relationships**.
5. Select **OK**.

This concludes defining the persistence layer for the one-to-one relationship.

Creating one-to-many (1-M) relationships

The Bank sample uses a one-to-many relationship for the *VapBankBranch* and the *VapAccount* business objects. This topic explains how to create a one-to-many relationship using the ObjectExtender tools.

Do the following:

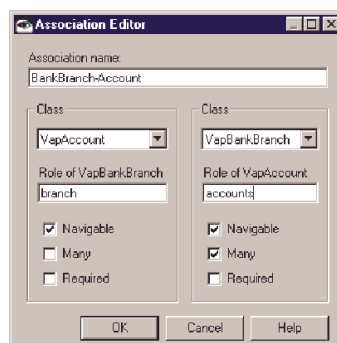
- Launch the Model Browser. Select **Bank** from the **Models** view, and *VapBankBranch* from the **Model Classes** view.

The **BankBranch-Account** class association is an example of a one-to-many relationship.

It was created as follows:

1. Select **New Association** from the **Associations** menu.

This launches the Association Editor.



2. In the **Association name** field, type **BankBranch-Account**.
3. Define the two **Class** sections.
 - a. In the left **Class** pane, select *VapAccount*.
 - b. In the right **Class** pane, select *VapBankBranch*.
 - c. Type **branch** for **Role of VapBankBranch**.
 - Select **Navigable**.

This means *VapBankBranch* can be obtained from the *VapAccount* object.

- d. Type **accounts** for **Role of VapAccount**.
- e. Select **Navigable**.

This means *VapAccount* can be obtained from the *VapBankBranch* object.

Select **Many**. This sets the cardinality to zero-to-many (0:M). If a cardinality of one-to-many (1:M) is desired, select **Required**.

4. Select **OK**.

You are now done with the Model Browser. The one-to-many relationship is defined.

Next, defining this relationship using schema semantics is shown. Finally, mapping the schema to the model is the last step for completing the persistence layer.

Do the following:

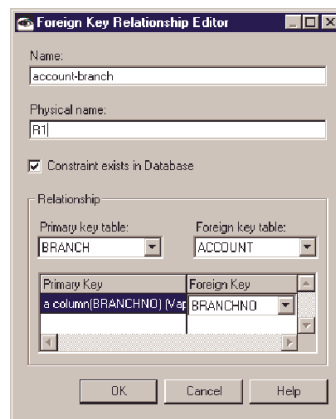
- Launch the Schema Browser, and select **Bank Sample** from the **Schemas** view, and select **BRANCH** from the **Tables** view.

The **account-branch** foreign key relationship is the one-to-many relationship that corresponds to the BankBranch-Account class association (relationship).

It was created as follows.

1. Select **Foreign Key Relationship** from the **Foreign_Keys** menu.

This launches the Foreign Key Relationship editor.



2. Type **account-branch** for the **Name**.
3. Type **R1** for the **Physical Name**.
4. If a foreign key constraint does not exist on the database, make sure **Constraint exists in database** is not selected.
5. Update the Relationship section:
 - a. Select **BRANCH** for the **Primary key table**.

The **Primary key** column (read-only) will be updated with the primary key from the table.
 - b. Select **ACCOUNT** for the **Foreign key table**.

In the **Foreign key** column, select the foreign key in the **ACCOUNT** table which corresponds to the branch, **BRANCHNO**.
6. Select **OK**.

You are now done with the Schema Browser.

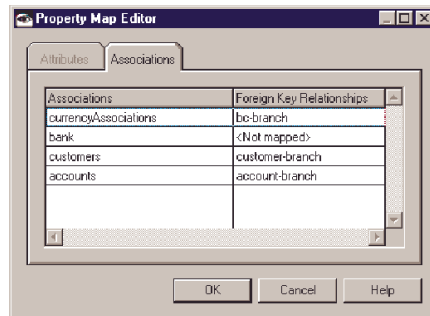
The last step in defining the persistence layer for the one-to-many relationship follows.

- Launch the Map Browser, and select **Bank Sample** from the **Datastore Maps** view, and select **VapBankBranch** from the **Persistent Classes** view, and then select **BRANCH** from the **Table Maps** view.

The **(r) accounts(account-branch)** property map represents the mapping from the **accounts** attribute from *VapBankBranch* to the **account-branch** foreign key relationship.

It was created as follows.

1. Select **Bank Sample** from the **Datastore Maps** view, **VapBankBranch** from the **Persistent Classes** view, and **BRANCH** from the **Table Maps** view.
2. Select **Edit Property Maps** from the **Table_Maps** menu.
This launches the Property Map Editor.
3. Click on the **Associations** tab.



4. Change the [Not Mapped] value of the **accounts** association to **account-branch** under **Foreign Key Relationships**.
5. Select **OK**.

This concludes the persistence layer for the one-to-many relationship.

Creating many-to-many (M-M) relationships

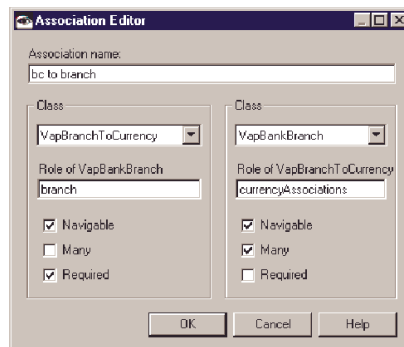
The Bank sample uses a many-to-many relationship for the *VapBrankBranch* and the *VapCurrency* business objects. With the current ObjectExtender implementation, you must construct a many-to-many relationship by joining together two one-to-many relationships: *VapBankBranch*-to-*VapBranchToCurrency* and *VapCurrency*-to-*VapBranchToCurrency*. This topic explains how to create a many-to-many relationship using the ObjectExtender tools.

Do the following:

- Launch the Model Browser. Select **Bank** from the **Models** view.
The **bc to branch** plus the **bc to currency** class associations is an example of a many-to-many relationship (As mentioned, the current implementation for a many-to-many relationship is to create two one-to-many relationships).

It was created as follows:

1. Select **New Association** from the **Associations** menu.
This launches the Association Editor.



2. In the **Association name** field, type **bc to branch**.
3. Define the two **Class** sections.

- a. In the left **Class** pane, select *VapBranchToCurrency*.
- b. In the right **Class** pane, select *VapBankBranch*.
- c. Type branch for **Role of VapBankBranch**.
- d. Select **Navigable**.

This means a *VapBankBranch* business object can be obtained from a *VapBranchToCurrency* business object. .

- e. Select **Required**. This sets a cardinality of one-to-one (1:1).
- f. Type currencyAssociations for the **Role of VapBankBranch**.
- g. Select **Navigable**.

This means a *VapAccount* business object can be obtained from a *VapBankBranch* business object.

- h. Select **Many**.

This sets a cardinality of zero-to-many (0:M). If a cardinality of one-to-many (1:M) is desired, select **Required**.

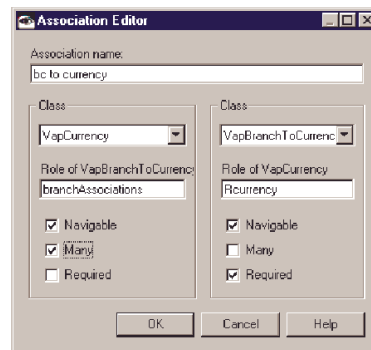
4. Select **OK**.

Next, you define the second of the two one-to-many relationships.

Do the following:

1. Select **New Association** from the **Associations** menu.

This launches the Association Editor.



2. In the **Association name** field, type bc to currency.
3. Define the two **Class** sections.
 - a. In the left **Class** pane, select *VapCurrency*.
 - b. In the right **Class** pane, select *VapBranchToCurrency*.
 - c. Type branchAssociations for **Role of VapBranchToCurrency**.
 - d. Select **Navigable**.

This means a *VapBankBranchToCurrency* business object can be obtained from a *VapCurrency* business object. .

- e. Select **Many**. This sets a cardinality of one-to-one(1:1).
- f. Type currency for the **Role of VapCurrency**.
- g. Select **Navigable**.

This means a *VapCurrency* business object can be obtained from a *VapBankToCurrency* business object.

- h. Select **Navigable**.
- i. Select **Required**.

This sets a cardinality of (1:1).

4. Select **OK**.

You are now done with the Model Browser. The many-to-many relationship is defined.

Next, defining this relationship using schema semantics is shown. Mapping the schema to the model will be the last step covered. These combined tasks provide the required information for creating persistence support for your business objects.

Do the following:

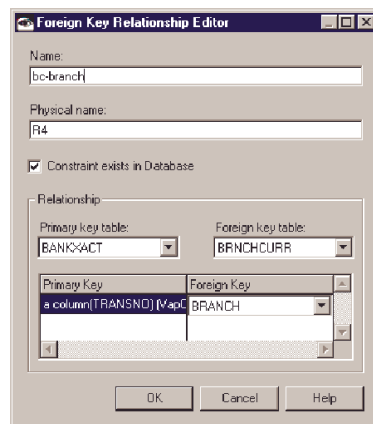
- Launch the Schema Browser, and select **Bank Sample** from the **Schemas** view, and select **BRNCHCURR** from the **Tables** view.

The **bc-branch** and **bc-currency** foreign key relationship is the many-to-many relationship that corresponds to the bc-to-branch and bc-to-currency class associations.

It was created as follows.

1. Select **Foreign Key Relationship** from the **Foreign_Keys** menu.

This launches the Foreign Key Relationship editor.



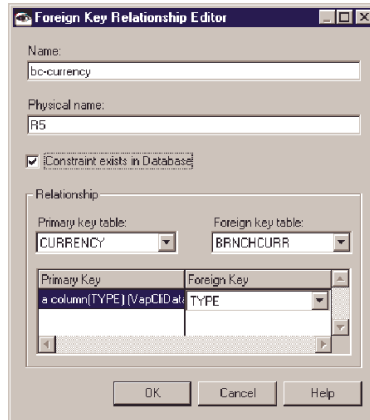
2. Type **bc-branch** for the **Name**.
3. Type **R4** for the **Physical Name**.
4. If a foreign key constraint does not exist on the database, make sure **Constraint exists in database** is not selected.
5. Update the Relationship section:
 - a. Select **BRANCH** for the **Primary key table**.
The **Primary key** column (read-only) will be updated with the primary key from the table.
 - b. Select **BRNCHCURR** for the **Foreign key table**.
In the **Foreign key** column, select the foreign key in the **BRNCHCURR** table which corresponds to the branch, **BRANCH**.
6. Select **OK**.

Next, you will define the **bc-currency** side of the foreign key relationship.

Do the following:

1. Select **Foreign Key Relationship** from the **Foreign_Keys** menu.

This launches the Foreign Key Relationship editor.



2. Type **bc-currency** for the **Name**.
3. Type **R5** for the **Physical Name**.
4. If a foreign key constraint does not exist on the database, make sure **Constraint exists in database** is not selected.
5. Update the Relationship section:
 - a. Select **CURRENCY** for the **Primary key table**
The **Primary key** column (read-only) will be updated with the primary key from the table.
 - b. Select **BRNCHCURR** for the **Foreign key table**.
In the **Foreign key** column, select the foreign key in the **BRNCHCURR** table which corresponds to the currency, **TYPE**.
6. Select **OK**.

You are now done with the Schema Browser.

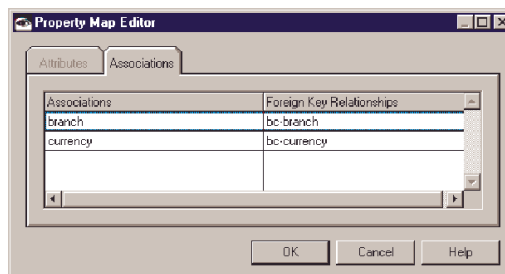
The last step in defining the persistence layer for the many-to-many relationship follows.

- Launch the Map Browser, and select **Bank Sample** from the **Datastore Maps** view, and select **VapBranchToCurrency** from the **Persistent Classes** view, and then select **BRNCHCURR** from the **Table Maps** view.

The **(r) branch (bc-branch)** property map represents the mapping from the **currency** attribute from *VapBranchToCurrency* to the **bc-currency** foreign key relationship.

It was created as follows.

1. Select **Bank Sample** from the **Datastore Maps** view, **VapBranchToCurrency** from the **Persistent Classes** view, and **BRNCHCURR** from the **Table Maps** view.
2. Select **Edit Property Maps** from the **Table_Maps** menu.
This launches the Property Map Editor.
3. Click the **Associations** tab.



4. Change the [Not Mapped] value of the **branch** association to **bc-branch** under **Foreign Key Relationships**.
5. Change the [Not Mapped] value of the **bc-currency** association to **currency** under **Foreign Key Relationships**.
6. Select **OK**.

This concludes defining the persistence layer for the many-to-many relationship.

Mapping business objects to tables

Classes are always mapped to at least one table. Secondary table maps allow the mapping of attributes to more than one table. ObjectExtender provides choices in how class hierarchies are mapped to table structures. There are basically two ways to match an inheritance tree to a relational database. Either you combine all attributes of the class hierarchy in one table, or you have a table for each class and establish the necessary foreign key relationships between the tables.

Creating a single table map with no inheritance

A table map with no inheritance is the most common type of mapping used. This type of map will be used to map attributes and relationships from one persistent object model class to one database table.

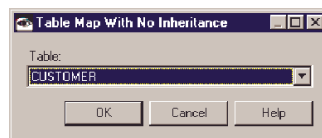
There are several examples of this type of mapping within the Bank sample model.

Launch the Map Browser, and select **BankSample** from the **Datastore Maps** view, and then select **VapCurrency** in the **Persistent Classes** view. In the Table Maps view, the **CURRENCY** table map listed is one example of a table map with no inheritance.

It was created as follows:

1. Select **VapCurrency** from the **Persistent Classes**.
2. Select **New Table Map** from the **Table_Maps** menu.

This opens the Table Map with No Inheritance editor.



3. Select the **CURRENCY** table from the **Table** list.
4. Select **OK**.

The table map has been created but there are no property maps for this table map. To map the attributes and associations for the table, select the **CURRENCY** table map and then select **Edit Property Maps** from the **Table_Maps** menu.

Creating a secondary table map

A secondary table map is necessary when a persistent model class is mapped to two or more database tables. An example of this type of mapping can be found in the Bank sample model.

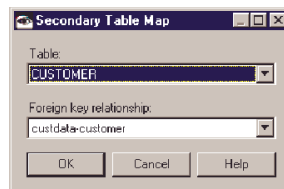
Launch the Map Browser, and select **BankSample** from the **Datastore Maps** view, and then select **VapCustomer** from the **Persistent Classes** view.

Note that two table maps are present for the **VapCustomer** persistent class. The **CUSTDATA** is the secondary table map.

It was created as follows

1. In the Bank sample schema, there needs to be a **CUSTDATA** and **CUSTOMER** table with all of the necessary columns added. It is the **CUSTDATA** table that is going to hold additional information for the **VapCustomer** persistent class.
2. Create a foreign key relationship named **custdata-customer** where **CUSTOMER** is the primary key table and **CUSTDATA** is the foreign key table. (Recall this is done using the Schema Browser).
3. In the Map Browser, select **VapCustomer**.
4. Create the **CUSTOMER** table map with no inheritance.

Note that when mapping the attributes in the Property Map Editor for this table, the **birthDate**, **felon**, **rating**, and **sex** will not get mapped because this table does not have the fields necessary to map these attributes. All of the associations can be mapped from this table.



5. From the **Table_Maps** menu, select **New Table Map - Add Secondary Table Map**.
6. Select the **CUSTDATA** table from the **Table** menu.
7. Select the **custdata-customer** relationship from the **Foreign Key Relationship** list.
8. Select **OK**.
9. Select the **CUSTDATA** table map.
10. Select **Edit Property Maps** from the **Table Maps** menu.
11. Select the corresponding table column for the class attributes: **birthDate**, **felon**, **rating**, and **sex**.
12. Select **OK**.

Creating single table inheritance maps

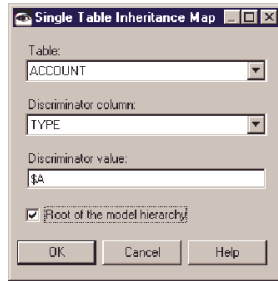
A single table inheritance map is used to map two or more persistent model classes to the same table. This is useful for persistent objects that have basically the same data but different behaviors. An example of this type of mapping is included with the Bank sample model.

Launch the Map Browser, and select **BankSample** from the **Datastore Maps** view, and then select **VapAccount** from the **Persistent Classes** view.

Note that **VapAccount** has two subclasses: **VapCheckingAccount** and **VapSavingsAccount**. All three of these classes map to the **ACCOUNT** table.

It was created as follows:

1. Select **VapAccount** from the **Persistent Classes** view.
2. From the **Table_Maps** menu, select **New Table Map - Add Single Table Inheritance Table Map**.



3. Select **ACCOUNT** for the **Table**.
4. Type: **\$A** for the **Discriminator value**.
5. Select **OK**.
6. Select **VapCheckingAccount** from the **Persistent Classes** view.
7. From the **Table_Maps** menu, select **New Table Map - Add Single Table Inheritance Table Map**.
8. Select **ACCOUNT** for the **Table**.
9. Type: **C** for the **Discriminator value**.
10. Select **OK**.
11. Select *VapSavingsAccount* from the **Persistent Classes** view.
12. From the **Table_Maps** menu, select **New Table Map - Add Single Table Inheritance Table Map**.
13. Select **ACCOUNT** for the **Table**.
14. Select **TYPE** for the **Discriminator column**.
15. Type: **S** for the **Discriminator value**.
16. Select **OK**.

The single table inheritance mapping is now defined.

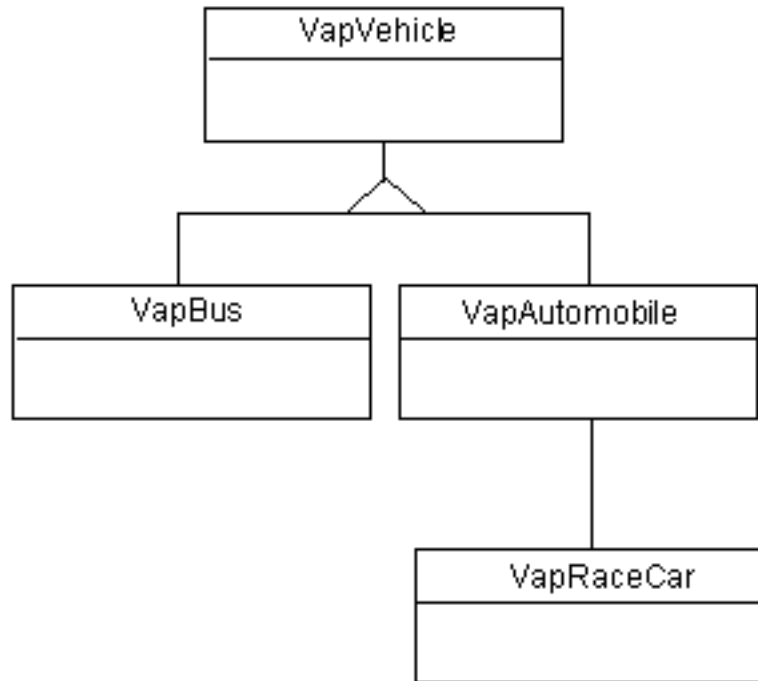
For a model that has inheritance, when a new schema is generated a single table inheritance approach is used. However, the table maps for the primary table is created once but added twice to each class map. To correct this, select one of the table maps and select the option in the browser to delete it.

Creating root/leaf inheritance maps

A root/leaf inheritance map is similar to the single table inheritance map, except that each subclass may have additional information to the superclass and this information will be stored in a separate table. When the subclass is instantiated, it will obtain its information from its table as well as from the superclasses' table. An example of this type of mapping can be found in the AutoWorld example.

Launch the Model Browser and select AutoWorld from the **Models** view. Double-click on **VapVehicle** to expand its hierarchy.

Below is a diagram of the class hierarchy for *VapVehicle* in the Model Browser.

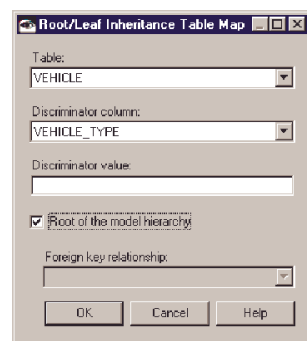


If you open the Schema Browser and select AutoWorld, you will notice that there is a separate table for each of the model objects in the *VapVehicle* class hierarchy (VEHICLE, BUS, AUTOMOBILE, RACECAR).

Each of the leaf tables (AUTOMOBILE, RACECAR, and so on) have foreign key relationships to their parent table. The important part of this example is how the models from the Model Browser are mapped to the tables in the Schema Browser. This can be seen in the Map Browser.

1. Launch the Map Browser.
2. Select AutoWorld from the **Datastore Maps** view.
3. Select **VapVehicle** from the **Persistent Classes**. Double-click on this class to expand its hierarchy.
4. From the **Table_Maps** menu, select **New Table Map - Add Root/Leaf Inheritance Table Map**.

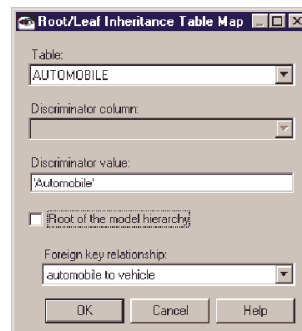
This opens the Root Leaf Inheritance Map editor.



5. Select VEHICLE for the **Table**.
6. Select VEHICLE_TYPE for **Discriminator column**.

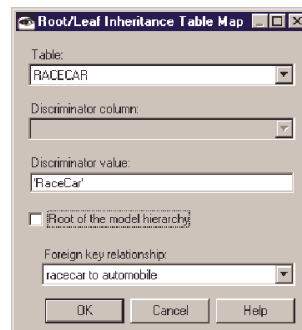
Note that a discriminator value is not entered for the VEHICLE table because the *VapVehicle* is only an abstract class and cannot be instantiated. If, in your application, the root of the model hierarchy can be instantiated, you must enter the discriminator value.

7. Ensure that the **Root of the model hierarchy** is selected.
8. Select **OK**.
9. Select **VapAutomobile** from the **Persistent Classes** view.
10. From the **Table_Maps** menu, select **New Table Map - Add Root/Leaf Inheritance Table Map**.



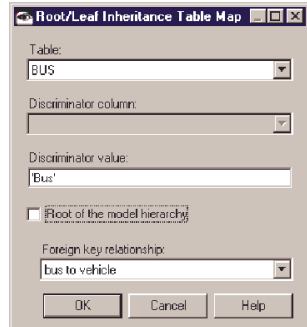
11. Select **AUTOMOBILE** for the **Table**.
12. Type **Automobile** for the **Discriminator value**.
13. Type **automobile to vehicle** for the **Foreign Key Relationship**.
14. Select **OK**.
15. Select **VapRaceCar** from the **Persistent Classes** view.
16. From the **Table_Maps** menu, select **New Table Map - Add Root/Leaf Inheritance Table Map**.

This opens the Root Leaf Inheritance Map editor.



17. Select **RACECAR** from the **Table** list.
18. Type **RaceCar** for the **Discriminator value**.
19. Select **racecar to automobile** from the **Foreign Key Relationship** menu.
20. Select **OK**.
21. Select **VapBus** from the **Persistent Classes** view.
22. From the **Table_Maps** menu, select **New Table Map - Add Root/Leaf Inheritance Table Map**.

This opens the Root Leaf Inheritance Map editor.



23. Select **BUS** from the **Table** list.
24. Type **Bus** for the **Discriminator value**.
25. Select **bus to vehicle** from the **Foreign Key Relationship** menu.
26. Select **OK**.

The root/leaf inheritance table map is now defined.

Using a composer for mapping an attribute to multiple database fields

A composer is used to map a single class attribute to multiple database columns. For example, in the Bank sample, the **name** for *VapCustomer* is mapped to three fields in the **CUSTOMER** table: **FIRSTNAME**, **MIDINIT**, and **LASTNAME**. If a composer does not exist that matches your data, you can create one by creating a new subclass of *VapCustomer*.

Note that composed attributes used as keys are not supported.

In this example, the *VapNameComposer* was created.

Do the following:

1. Create a new class, *VapNameComposer*, that is a subclass of *VapComposer* *VapAttributeComposer*. Next, implement the following methods:
 - *sourceDatatype*
 - *targetClass*
 - *attributeNames*
 - *objectFrom:*
 - *dataFrom:*
2. Implement an instance method called *targetClass* (The target class in this example will be *VapName*). This should return the name of the class for the instance created as a result of the *objectFrom:* message sent to the converter. Note that in some cases it may be desirable to create your own target class such as was done in this example.
3. Implement an instance method called *attributeNames* . It should return an array of the attribute name strings from the target class. For example:

```
^#('firstName' 'middle' 'lastName')
```

4. Implement an instance method called *objectFrom:* . The argument will be an array containing the values for the attribute string name from the method *attributeNames* in the same order. This method will set the target class attributes based on these values and return an instance of the target class. For example:

```
^VapName first: (array at: 1) middle: (array at: 2) last: (array at: 3))
```

5. Implement an instance method called *sourceDatatype* . This method should return an array of the data elements' class names passed as a parameter to the *objectFrom:* message which is sent to the converter.

For example,

```
^(String String String)
```

6. Implement an instance method called *dataFrom:* . The argument will be an instance of the target class. This method is responsible for returning a collection of objects that are to go to data store fields. For example:

```
^(Array
  with: anObject firstName
  with: anObject middle
  with: anObject last)
```

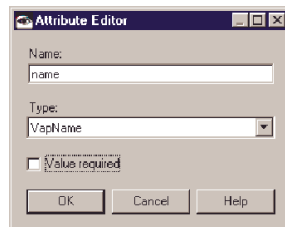
The *VapName* class is used in the Bank sample to map one attribute to several columns. It was used as follows:

1. Launch the Model Browser.
2. Select Bank from the **Models**.
3. Select *VapCustomer* from the **Model Classes**.
4. Select **Edit Class** from the **Classes** menu.

This opens the Class Editor.

5. Click **New** to add a new attribute.

This opens the Attribute Editor.



6. Type name in the **Name** field.
7. Select *VapName* from the **Type** list.

VapName appears in the Type list because the new subclass of *VapComposer*, *VapNameComposer*, was recognized with *VapName* as its target type. The target type of any *VapComposer* subclasses you create will also appear here.

8. Select **OK**.

This concludes the example.

Changing part of a composed attribute like name does not cause the Customer object to be marked "dirty" and updated in the database. To make the Customer object dirty because of a name change, you must alter the name attribute through the Customer's setName accessor.

Also note that if you have a complex attribute type and you want to change the composer type, you must delete the complex attribute map from the Map Browser and recreate it in the Property Map Editor.

Using converters

There may be times when you need to convert data being read from the database and being stored to the database. An example of this would be converting a

character to a Boolean. In the database, a value may be stored as 'Y', but when it is retrieved, it may be converted to the Boolean object **true** and vice versa. There are many different types of converters that may be used, for example, *VapCharToBoolean*, *VapCharToString*, and so on. You may also create your own converter by subclassing from *VapAbstractConverter*.

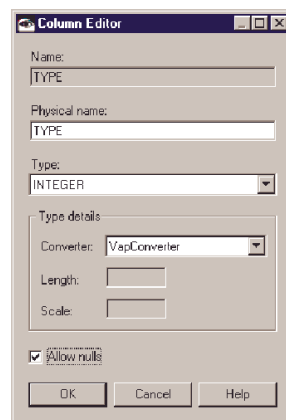
The steps for using a converter are as follows:

1. Launch the Schema Browser.
2. Select a schema from the **Schemas** view.
3. Select a table from the **Tables** view.
4. Select **Edit Table** from the **Tables** menu.

This opens the Table Editor.

5. Select an existing column to edit from the **Table columns** view, and click on **Edit**.

This opens the Column Editor.



6. Select the desired converter from the **Converter** list.
7. Select **OK**.

Various vendor database drivers treat single-character data differently. Some will return a Character, some will return a String of length one. To be sure, if an application may run with multiple drivers, type Char(1) fields as Strings, and use the *VapCharToString* converter. It will guarantee that a String will always be returned.

Performance tuning

Changing the locking type on a table

You can selectively enable the pessimistic locking policy for the classes that are being mapped to the data store. By default the pessimistic locking is not enabled for any of the classes in the model.

To enable pessimistic locking, do the following:

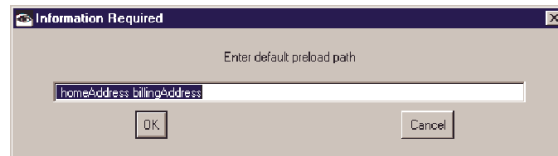
1. Launch the Map Browser.
2. Select the desired map from the **Datastore Maps** view.
3. Select the desired class from the **Persistent Classes** view.
4. Select **Enable pessimistic locking** from the **Persistent_Classes** menu.

If there is a class hierarchy, and pessimistic locking is desired, it must be enabled in the subclass as well.

Setting preload paths

To set a default preload path for a class, do the following:

1. Launch the Map Browser.
2. Select the desired map from the **Datastore Maps** view, for example, Bank Sample.
3. Select the desired class from the **Persistent Classes** view, for example, **VapCustomer**.
4. Select **Change default preload path** from the **Persistent_Classes** menu.
5. Enter the attribute names (separated by spaces) that will be instantiated when the object is first read from the database, for example, homeAddress billingAddress.



6. Select **OK**.

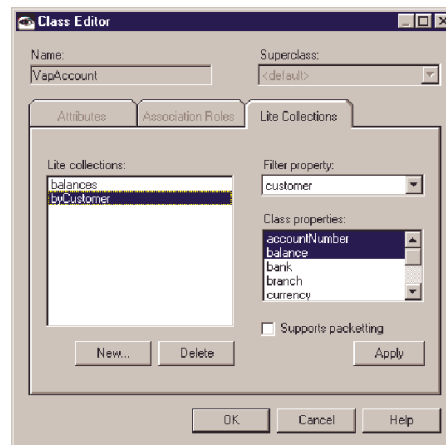
Creating Lite collections

Lite Collections are useful for retrieving a subset of the information from a particular object in the database without instantiating each object that is retrieved. An example of the use of lite collections can be found within the Bank sample model. Two lite collections were created for the VapAccount class.

The lite collections were created as follows:

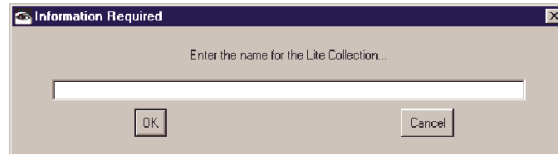
1. Launch the Model Browser.
2. Select Bank from the **Models** view.
3. Select *VapAccount* from the **Model Classes**.
4. Select **Edit Class** from the **Classes** menu.

This launches the Class Editor.



5. Click on the **Lite Collections** tab.
6. Click **New**.

This launches a dialog.



7. Type byCustomer for the name of the lite collection.
8. Select **OK**.
9. Select **Customer** from the **Filter property** menu.
10. Select **balance** and **accountNumber**, the properties to be retrieved, from the Class properties.
11. Select **Supports Packeting** if you want packeting turned on.
12. Select **Apply**.
13. Select **OK**.

Chapter 6. Reference

This section covers the following topics:

- “ObjectExtender runtime architecture”
 - “Programming model overview”
 - “The business object layer” on page 90
 - “The persistence layer” on page 102
- “Metadata storage model” on page 108
- “Writing your own services” on page 110
- “User code sections” on page 110
- “Managing business objects” on page 111

ObjectExtender runtime architecture

ObjectExtender provides transactional support and backend store mapping for objects and relationships. The runtime system is separated into two major layers:

- **Business object layer.** The business object layer provides support for business objects and the management of relationships between them. It also provides transaction isolation (including support for nested transactions) for changes made to business objects and relationships. The entire business object layer including transaction and relationship support can run without the persistence layer.
- **Persistence layer.** The persistence layer provides support for mapping objects to and from legacy backends. Backends supported include relational databases and “function call” backends.

Programming model overview

ObjectExtender offers a single-level store model. Tracking which objects are created, deleted or modified in a transaction is done by ObjectExtender. Thus, at the highest level, the program flow is:

1. Start a transaction.
2. Create, Retrieve, Update and Delete objects, Navigate, Add and Remove from relationships.
3. Commit or roll back the transaction.

ObjectExtender offers support for both optimistic (non-locking) and pessimistic (locking) policies for managing transactions. In the case of optimistic management, ObjectExtender defers all locking and updates on the backend store to the commit phase of the ObjectExtender transaction. This means that, at least in the case of optimistic transaction management, it is feasible to have long-running ObjectExtender transactions that have good concurrency characteristics on the backing store, because they do not consume locks or other significant backend resources. In this case, the programming model looks more like:

1. Start a transaction.
2. Repeatedly interact with a user and Create, Retrieve, Update and Delete objects, Navigate, Add and Remove from relationships.
3. Commit the transaction.

ObjectExtender supports nested transactions, so a more complete model would be:

1. Repeatedly interact with a user and Retrieve objects, Navigate relationships.
2. Start a transaction, *t1*.

3. Repeatedly interact with a user and Create, Retrieve, Update and Delete objects, Navigate, Add and Remove from relationships.
4. Create a nested transaction, *t2*.
5. Repeatedly interact with a user and Create, Retrieve, Update and Delete objects, Navigate, Add and Remove from relationships.
6. Commit or roll back the nested transaction *t2*.
7. Repeatedly interact with a user and Create, Retrieve, Update and Delete objects, Navigate, Add and Remove from relationships.
8. Commit the top-level transaction *t1*.
9. Repeatedly interact with a user and Retrieve objects, Navigate relationships.

There is no requirement that a transaction complete before a thread creates a new sibling transaction. For example, a thread may start a transaction, modify some objects, suspend the transaction and create another, modify more objects, resume and commit the first transaction and so on. There is also no requirement that nested transaction complete before its parent resumes. Within a transaction, modifications made in a parent transaction are visible, but modifications made in uncommitted sibling or child transactions are not.

ObjectExtender also supports independent Transactions executing simultaneously on separate threads, so the above program flows can be running simultaneously on multiple threads of execution within the same Smalltalk program. Nested transactions must always run on the same thread as their parent.

The business object layer

The business object layer is implemented in part by a small runtime and in part by a small set of interface and behavior requirements for business objects. Code generation support is supplied (in the ObjectExtender tool set) to generate code (and selectively update previously-generated code) for business object and relationship classes from a high-level description of the classes and their relationships.

Business objects and their relationships are described using an extended entity relationship model using a UML vocabulary. Supported features include business object inheritance and relationship cardinality, navigability and inverses. Business objects are not required to inherit from a common root, although an abstract class with suitable behaviors is provided for convenience. Separate classes are generated for relationships to allow business rules to be written governing relationships. In addition to the business object classes that are generated, *Key* classes and *HomeCollection* classes are generated for each business object.

New methods can be freely added to business objects using normal browsers without any need to regenerate any part of the runtime system. Only when structural changes are made to objects (adding, deleting or changing an attribute or relationship definition) is there any requirement to regenerate. Non-managed fields can also be added without regeneration.

HomeCollections

ObjectExtender generates a *HomeCollection* class for each business object class. The protocol on *HomeCollection* classes is the one defined by Component Broker (*createFromKey*:, *findByPrimaryKey*:).

ObjectExtender does not prescribe how the *HomeCollections* are located by the application code. A default mechanism is supplied that will store a singleton *HomeCollection* instance in a static variable of each *HomeCollection* class, and this mechanism may be used by application code to locate appropriate *HomeCollection*

instances. However, the ObjectExtender framework never uses this access path, and application programmers are free to store and locate *HomeCollection* instances in other ways.

HomeCollections also have the following responsibilities:

- Knowing the *DataStore* they are attached to.
This is a responsibility of *HomeCollections* for persistent business objects only. A *DataStore* represents a particular backend store (for example, a database or a set of CICS TPs) that stores objects from a model. The *HomeCollection* provides the mapping of a set of classes to a data store.
- Knowing the service implementation to be used for all instances from this home.
In fact, the *HomeCollection* is the object that maps the business object layer to a particular *DataStore* and service object implementation. *HomeCollection* classes for persistent objects subclass from a different class from *HomeCollection* classes for non-persistent classes. The persistent *HomeCollection* classes have extra protocol for dealing with this mapping of objects to a backend.
- Knowing the isolation policy implementation for the *BusinessObject* class.
This means understanding whether objects in this home use optimistic (non-locking) or pessimistic (locking) transaction isolation, and providing an appropriate implementation object for the same.
- Knowing about other related homes for the same object model on the same data store.
A *HomeCollection* knows about the home for its business object's superclass and also about the homes for its business object's subclasses. It also knows about the homes for all of the relationships from its business object. (Relationships have homes as well).

Transaction overview

The transaction model supports **concurrent** and **nested** transactions.

A nested transaction is a tree of transactions. The subtrees can be flat or nested transactions. At the leaf level, transactions are flat. The root of the tree is the **top-level transaction**; all others are **subtransactions**. A transaction's predecessor in the tree is a **parent**; a subtransaction at the next lower level is a **child**. A subtransaction can either commit or roll back; its commit will not take effect, unless the parent transaction commits. Therefore, any subtransaction can finally commit only if the top-level transaction commits. The rollback of a transaction anywhere in the tree causes all its subtransactions to roll back.

The commit of a subtransaction makes its results accessible only to the parent transaction. All objects held by a parent transaction can be made accessible to its subtransactions. Changes made by a subtransaction are not visible to its siblings, in case they execute concurrently.

There is always at least one active transaction: a global read-only transaction, which is called the **shared transaction**. When a new (read/write) top-level transaction is created, it becomes a child of the shared transaction. When there are multiple transactions, the application must explicitly set which of the transactions is the current one. All the modifications to the business objects are recorded by the **current transaction**.

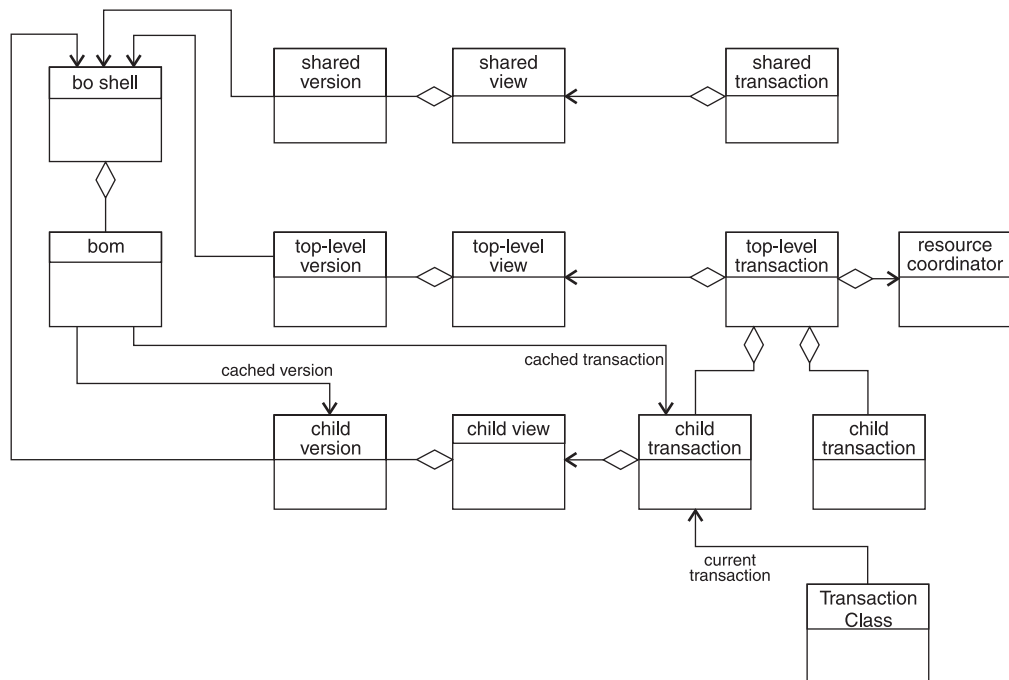


Figure 17. Transaction instance coordination. The UML diagram shows the coordination of the shared, top-level, and child transaction instances .

Each transaction has its own view. A view is snapshot (of a subset) of the application's business object model. Each business object is divided into two parts: a **shell** and a **version**. An object's business behavior is in the shell, and the instance data is in the version. When a business object is first accessed (get/set a property) within a transaction, a new version of the object is added to the current transaction's view. The new version is based on the version in the parent transaction's view (if the parent transaction's view does not contain a version of the object, the parent will first create a version for itself based on its parent). When any object refers to a business object, it actually refers to the business object's shell. The shell dynamically connects itself to the current version in the current transaction's view. This way the shell/version pair implements a **dynamic reference** to a business object.

Each transaction has a **resource coordinator**. The resource coordinator's responsibility is to ensure that the modifications made within the transaction become persistent across multiple resources. A resource provides the ACID properties for a particular connection to a data store.

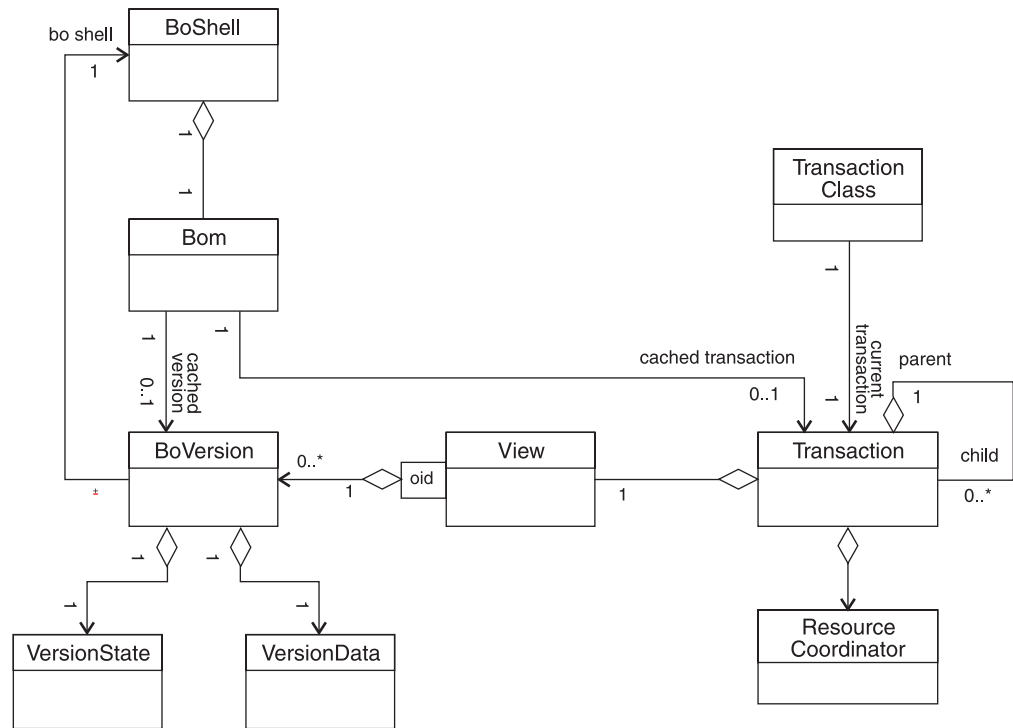


Figure 18. Transaction/View/Version flow. The UML diagram shows the flow of the Transaction/View/Version model.

On commit, a transaction's view is merged to its parent transaction's view (the top-level view is merged to the shared view), and the view and the resources are synchronized. When receiving the commit request, the transaction first tests if its view can be merged to its parent's view. The default test is that if the same object has been modified in both parent and child views, the views cannot be merged. If the test fails, the transaction will be rolled back. If the test succeeds, the transaction requests its resource coordinator to synchronize the view and the resources. The resource coordinator passes each version in the view to a corresponding resource. The coordinator then prepares and commits the changes to the resources. For nested transactions, the coordinator includes only resources that support nesting (if none of the resources supports nesting, the nesting happens only within the application memory). If the synchronization fails, the transaction will be rolled back. If the synchronization succeeds, the transaction's view is merged

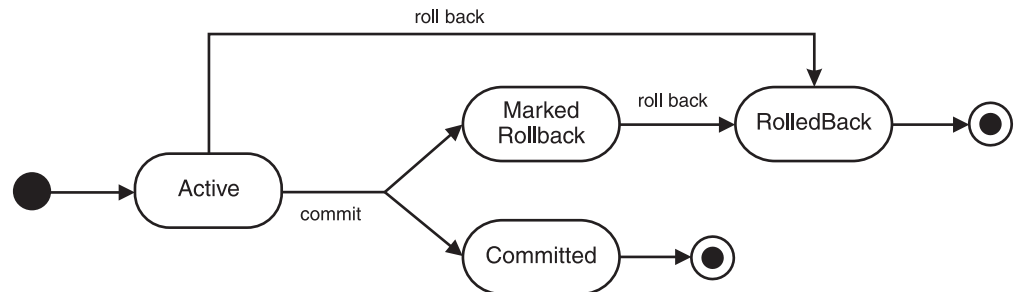


Figure 19. Transaction states. The UML diagram depicts the flow of the Transaction states.

with the parent view and the transaction is marked to be committed.

Transaction isolation policies

ObjectExtender has a *Transaction* class. Transactions represent paths of code execution. You access and manipulate data in your business objects within transactions.

New transactions are created by invoking the static method *begin* on *Transaction*. This always creates a new top-level transaction. To create a nested transaction, you send *beginChild* to an existing transaction. Each thread of execution has a notion of a current transaction. You can get the current transaction for the current thread of execution by sending *current* to *Transaction*.

Transaction Isolation: ObjectExtender manages isolation of changes between transactions. Isolation of changes between transactions means that outside the scope of a transaction, changes made within the transaction will not be seen until the transaction commits.

ObjectExtender supports both optimistic and pessimistic approaches to implementation of transaction isolation.

ObjectExtender implements transaction isolation according to the policies for the transaction and for the (service implementations for) *HomeCollections*.

The *Transaction* specifies either **Repeatable Read** or **Unrepeatable Read** isolation level.

The service implementation for the *HomeCollection* specifies either **non-locking** or **locking** implementation. The service objects therefore have implementation for:

- Non-locking Repeatable Read
- Non-locking Unrepeatable Read
- Locking Repeatable Read
- Locking Unrepeatable Read

Any given service implementation is required only to contain implementations of an non-locking or locking set, because the non-locking/locking flag is not changeable for a service. The locking capability for a class is specified by checking or unchecking the **Enable pessimistic locking** menu item for a persistent class in the Map Browser. The transaction isolation policy is specified by calling the methods *supportRepeatableReads()* or *supportUnrepeatableReads()* on a *Transaction*.

Repeatable Read: This is an isolation policy value that can be specified on a transaction-by-transaction basis. **Repeatable Read** guarantees that if the same object is fetched multiple times within the same transaction, (for example, using *findByPrimaryKey()*), then the fetched object will have the same attribute values each time. It is the rough analog of the DB2[®] read stability isolation level

Unrepeatable Read: This is an isolation policy value that can be specified on a transaction-by-transaction basis. **Unrepeatable Read** guarantees that if a business object is used within a transaction, then the attribute values of the business object within a transaction will not be affected by uncommitted changes to the business object in sibling transactions. However, if a sibling transaction commits before the current transaction, changes made in the sibling may become visible in the current transaction. This is the rough analog of the DB2 cursor stability isolation level.

Non-locking Implementation of Repeatable Read (Copy on read): This is an implementation of **Repeatable Read** specified on a type-by-type basis that uses copying. A copy of the data of each business object (including its associated data

object) will be made the first time it is read within a transaction. Within this transaction, subsequent read or write access to the object, or navigation to the object, or query returning the object will use the data from this copy.

Non-locking Implementation of Unrepeatable Read (Copy on write): This is an implementation of **Unrepeatable Read** specified on a type-by-type basis that uses copying. A copy of the data of each business object (including its associated data object) will be made the first time it is updated within a transaction. (Read-only access prior to the first write will use the shared copy of the data for the BO.) Within this transaction, subsequent read or write access to the object, or navigation to the object, or query returning the object will use this copy. Copies are not made on read. This means that if an application reads an object, and later re-reads the object (or reuses a stored reference to the object) it may see changed attribute values. In the current implementation, if you keep a reference to the object, you will only see changes made by sibling committed transactions that execute within the same process. If no reference to the object is held, you may see changes committed by transactions executing in other processes. However, we do not preclude the possibility that changes made by sibling transactions in other processes will be visible in the future even if a reference to the object is held.

Locking Implementation of Repeatable Read (Lock on read): This is an implementation of **Repeatable Read** specified on a type-by-type basis that uses locking. An object-level shared lock is acquired on each business object the first time it is read within a transaction. All subsequent attempts to acquire an update lock on this object in other transactions (except child transactions of the current transaction) will cause blocking or an exception. If the object is subsequently updated within the transaction, the lock will be upgraded to an update lock. In the persistence layer (see later) locks will also be acquired on the underlying data store to block other writers (if the object has only been read) and readers (if the object has also been updated) outside the current process. One option that can be specified for locking is not to procure a lock at all. This is typically used when it is known that the access patterns of the application require procurement of a lock on another object first. For example, it might be possible to specify no locking on *LineItems* if the only way to get to *LineItems* is through the owning *Invoice* object (which would be locked). Note that this is not the same as non-locking. Non-locking says that we expect conflicts on an object, but will detect and resolve these conflicts at commit time. Locking says that we expect no conflict.

Locking Implementation of Unrepeatable Read (Lock on write): This is an implementation of **Unrepeatable Read** specified on a type-by-type basis that uses locking. A lock is acquired on each business object the first time it is updated within a transaction. All subsequent attempts to acquire a write lock on this object in other transactions (except child transactions of the current transaction) will cause blocking or an exception. In the persistence layer locks will also be acquired on the underlying data store to block other writers and readers outside the current process. In the current implementation, if you keep a reference to the object, you will only see changes made by sibling committed transactions that execute within the same process. If no reference to the object is held, you may see changes committed by transactions executing in other processes. However, we do not preclude the possibility that changes made by sibling transactions in other processes will be visible in the future even if a reference to the object is held. One option that can be specified for locking is not to procure a lock at all. This is typically used when it is known that the access patterns of the application require procurement of a lock on another object first. For example, it might be possible to specify no locking on *LineItems* if the only way to get to *LineItems* is through the owning *Invoice* object (which would be locked). Note that this is not the same as

non-locking. Non-locking says that we expect conflicts on an object, but will detect and resolve these conflicts at commit time. Locking says that we expect no conflict.

Transaction API

Transaction provides the following class protocol:

#begin

Begin an anonymous top-level transaction.

#begin: aName

Begin a named top-level transaction (name is used only for printing and debugging).

#current

Answer the current transaction.

#currentView

Answer the current transaction's view.

#reset

Discard all transactions.

#resume

Set the current transaction.

#shared

Answer the shared transaction.

#suspend

Suspend the current transaction.

#topLevelDo:

Evaluate the block in the context of a new top-level transaction which is then committed. Ensure that the current transaction before this method is executed resumes itself as the current transaction.

Transaction provides the following instance protocol:

#beginChild

Begin an anonymous child transaction.

#beginChild: aName

Begin a named child transaction.

#commit

Request a transaction to commit. The application must handle the potential failure exceptions.

#commitOrRollback

Request a transaction to commit. In case of failure rollback the transaction. Return a Boolean indicating success, true means committed, false means rolled-back.

#commitWhenFailureDo: aFailureBlock

Request a transaction to commit. On failure execute the failure block.

#commitWhenSuccessDo: aSuccessBlock

Request a transaction to commit. On success execute the success block.

#commitWhenSuccessDo: aSuccessBlock whenFailureDo: aFailureBlock

Request a transaction to commit. On success execute the success block, and on failure execute the failure block.

#evaluate

Execute a block within the transaction.

#isActive

Test if the transaction is active.

#isChild

Test if the transaction is a child transaction.

#isCommitted

Test if the transaction is committed.

#isDescendantOf:

Test if the transaction is descendant of another transaction.

#isMarkedRollback

Test if the transaction is marked to be rolled back.

#isolationPolicy

Answer transaction's isolation policy.

#isolationPolicy:

Sets the transaction's isolation policy.

#isRolledBack

Test if the transaction is rolled back.

#isShared

Test if the transaction is a shared transaction.

#isTopLevel

Test if the transaction is a top level transaction.

#name Answer the transaction's name.**#name:**

Set the transaction's name.

#parent

Answer child transaction's parent transaction.

#resume

Resume the transaction (make it the current one).

#rollback

Request a transaction to rollback.

#supportDirtyReads

Sets the isolation policy to support dirty reads.

#supportRepeatableReads

Sets the isolation policy to support repeatable reads.

#supportUnrepeatableReads

Sets the isolation policy to create unrepeatable reads.

#suspend

Suspend the transaction (if current, reset the current).

Setting the transaction's isolation policy

This section contains a collection of code examples for using the collision management policies for your transaction layer.

Isolation policy: tries to lock objects. To set the transaction's isolation policy such that it tries to lock objects, use the following API.

```
tx1 supportRepeatableReads "this is the default"
```

Isolation policy: do not try to lock objects. To set the transaction's isolation policy such that it does not try to lock objects:

```
tx1 supportUnrepeatableReads
```

Timing: when the objects lock. If the transaction supports repeatable reads, the locking-capable objects are locked when

- They are touched, that is,

```
faculty name: 'Dr. Salo'
```

- They are read from the database using *allInstances* or *findByPrimaryKey*.

Handling exceptions. In the above cases, the application must be prepared to handle the *ExVapObjectLocked* exception, which is raised when the attempt to acquire the lock fails.

```
tx1 := Transaction begin.  
[depts := VapDepartment singleton allInstances. "..."]  
  when: ExVapObjectLocked  
  do: [:aSignal | "notify the user that the object is locked"]
```

Explicit locking. An object can be explicitly locked regardless of the transaction's locking policy.

```
[faculty lock]  
  when: ExVapObjectLocked  
  do: [:aSignal | "notify the user that the object is already locked"]
```

Explicit refresh. An object can be explicitly refreshed from the database.

```
faculty refresh
```

Collision detection predicates. If the object has collision detection predicates, the transaction fails when the object is updated in the database.

```
faculty := VapFaculty singleton findByPrimaryKey: aKey.  
faculty name: 'Dr. Rich'.  
tx1  
  commitWhenFailureDo:
```

```
[aSignal | tx1 rollback.  
"notify the user that there is a collision"]].
```

The failed object is always the first argument of the signal.

```
[aSignal | aBo := "aSignal argument ..."]
```

Depending on the exception the signal may have additional arguments, like the native SQL error.

Object uniqueness

Every business object defined to `ObjectExtender` must be identifiable by a unique key. One of the features of `ObjectExtender` is that it guarantees that only one instance of an object will exist within a transaction scope for a particular unique key. Each transaction keeps a registry of objects that have been read or written within that transaction. Whenever an object is requested, either in a query or by navigating a relationship, `ObjectExtender` will check if this object is already in the registry of this transaction or one of its parents. If it is, the object in the registry will be used.

Shared transaction

For client applications with a UI, it is very important to manage reads of objects outside of any transaction scope. This is not just a matter of convenience, but of semantics.

The model that `ObjectExtender` supports is as follows:

- The application can navigate through data, populating UI lists "outside of transaction scope".
- When you modify an object or relationship, the application should begin a top-level transaction.
- When a top-level transaction commits, the changes are committed to the external backing store and also applied to the objects that are "outside of transaction scope".

To support this model, `ObjectExtender` supports the notion of a special read-only **Shared Transaction** that is the parent of all top-level transactions.

Note that this model cannot be supported using simply top-level and nested transactions, because the model depends on the fact that the level of transaction that commits to the external store is the level above the Shared Transaction. In `ObjectExtender`, only top-level transactions commit to backing stores.

The *SharedTransaction* always runs with a **Unrepeatable Read** isolation policy. This allows the transaction registry to be weak, so that objects read in the shared transaction are not held onto after the last application reference to them is gone.

For transactions whose isolation policy is **Unrepeatable Read**, the registry is a weak structure. This means that if there are no other references to the object in the image, it may be reclaimed by the garbage collector. If the transaction isolation policy is **Repeatable Read**, the registry structure is strong. This guarantees that if the same query is executed twice in succession, the same objects with the same

attribute and relationship values will be returned the second time, even if no reference to the objects were kept in between queries.

Weakness is only implemented in the Smalltalk version of the product at this time.

The "shell" business object

Within the scope of a single transaction, all references to a business object of a particular key value are guaranteed to point to the same business object instance. Since the transaction may have numerous nested transactions, and since ObjectExtender supports optimistic (non-locking) transaction isolation, this means that ObjectExtender has to manage different values of the data of this business object in different nested transaction scopes.

In fact, because of the outer shared transaction, each top-level transaction is in fact a nested transaction for this purpose. This means that the same business object (BO) key value is resolved to reference the same BO instance, even across top-level transactions, so this mechanism is required for the simple non-nested top-level transaction case as well as for the nested case.

ObjectExtender implements this capability of having different values for the data of a single instance of an object in different transactions by generating special get and set methods that must be used to access the fields of an object. When an object is first touched within a transaction, a copy of the values of all the managed fields of the object is taken and stored associated with the transaction. The generated get and set methods will always return the value of the fields from the copy held in the transaction.

Some important characteristics of the way this works follow:

- There is no requirement that the class of the hidden copies of the data for business objects be the same as the class for the business objects themselves (by default they are, currently). The only protocol on the hidden copies of the state for business objects is *primGetSomeAttribute* and *primSetSomeAttribute*.
- The hidden copies of the data for business objects used for optimistic (non-locking) transaction isolation are completely transparent to the application programmer and the business object programmer.
- Pointers in the hidden copies of the data for business objects never point to copies of data. They always point back to the original business object instance.
- We call the original business object instance the **"shell" business object**, to distinguish it from the copies whose sole purpose is to store the field values for a particular transaction.
- All business object methods are executed on shell business objects. User logic never executes on the hidden copies of the state for business objects.
- All access to managed instance variables for a business object is required to go through generated get and set methods. Currently there is no check for violations of this rule. The reasons for this rule are:
 - The generated get and set methods contain special logic to get and set values into the correct copy for the current transaction.
 - The generated set methods contain special logic to register the object as modified in the current transaction.
 - The generated get and set methods contain special logic to update and de-reference relationships.
- The special variable, **super**, can be used without special consideration because it always (correctly) points to a shell business object.

Event Signaling

An overview of the ObjectExtender event signaling follows.

ObjectExtender supports the registration of listeners and signaling of events on business objects and relationships.

A code-generation object will cause all ObjectExtender-managed attributes to be created as attributes.

Listener registration is transaction-sensitive in ObjectExtender. Whenever a listener is registered for an object within a transaction, ObjectExtender will create a listener list within the transaction for that object if it has not already been created and add the listener to the list.

The initial listener list for a *propertyChanged* event in a transaction is always empty, even if the parent transaction had a non-empty property list. This means that listeners in the parent transaction will not be (immediately) notified of changes that happen within a child transaction.

ObjectExtender will signal a changed event whenever a property value is signaled. Only listeners that were added within the current transaction will be notified.

Whenever a transaction is committed, its modified objects are promoted to its parent, and any resulting changes to attributes will trigger a corresponding *propertyChanged* notification. This notification will go to listeners registered in the parent transaction.

ObjectExtender *LinkCollections* for relationships signal changes for adding and removing objects within them. ObjectExtender *LinkCollections* also signal an *elementChanged* event whenever one of their elements signals a *propertyChanged* event. This allows containers that display details of elements within the collection to get notifications of changes without having to add themselves as *propertyChanged* listeners of every element of the relationship.

Whenever a change to a *LinkCollections* is promoted to a parent transaction as part of transaction commit, a suitable sequence of add and remove events is signaled in the parent transaction.

Whenever a property of an object is modified, ObjectExtender will not only signal the *propertyChanged* event for the object itself, but will also signal a corresponding *elementChanged* event for every relationship that the object participates in.

BOManagers, Versions, and VersionStates

ObjectExtender requires each "shell" business object (BO) to be able to return a *BOManager* object.

The *BOManager* object has the following responsibilities:

- Remember the HomeCollection for the BO.
- Remember the key for the BO.
- Create a temporary key for new objects that do not yet have one.
- Return the read or write version for the BO in the current transaction, creating it if it does not already exist.
- Hold the lock object for a BO.
- Cache the most recently used transaction version for rapid access.

In order to track the state and data values of an object within a particular transaction, ObjectExtender creates *Version* and *VersionState* objects for each object that is read or updated within a transaction. The *Version* objects are the objects that are actually stored in the transaction registries, keyed by BO key. (Recall that it is the transaction registries that are used to implement object uniqueness within a transaction and to track which objects are "dirty" in a transaction). *Version* objects are also cached by *BOManagers* for rapid repeat access within the same transaction.

A *Version* object has the following responsibilities:

- Remember the transaction the version is in.
- Remember the "shell" BO for the version.
- Remember the hidden copy of the data for the BO.
- Remember the last modification count of the version. This count is bumped every time the version is updated.
- Remember the last modification count of the parent version at the time the version was made. This value is used to know whether the version of the BO in the parent transaction has been modified subsequent to this version being created. This is used to collect optimistic (non-locking) transaction isolation collisions.
- Remember the state of the version. This is encapsulated in a separate state object. The basic states are
 - initial
 - new
 - retrieved
 - deleted.
- Handle state transitions. For example, objects may go from retrieved to modified or deleted.
- Detect transaction isolation collisions (delegated to the state objects and isolation policies).
- Acquire and release locks (requested by transaction isolation policy, performed when a version is first created for read or write, delegated to isolation implementor).
- Register version in transaction registry.
- For persistent BO versions perform the appropriate Create/Read/Update/Delete (CRUD) on the database based on version state to reflect changes. This is driven by the resource object during transaction commit, and is interpreted by the versions helper state object.

VersionState objects are helper objects for version objects.

The persistence layer

In addition to the support for objects, relationships and transactions provided by the business object layer, ObjectExtender provides support for persistence by mapping objects to a database or "function call" backend.

The persistence layer is implemented by generating parallel class hierarchies of classes to the *business object hierarchy*. Instances of these parallel classes are responsible for mapping the business objects and relationships to and from the backend store.

Two of these hierarchies are:

- *DataObject*
- *ServiceObject*.

DataObjects

Data objects contain the data for a business object in the form in which it was retrieved from the persistent store. Each persistent business object points to a data object, and there is a direct correspondence between the managed fields of the business object and the fields of the data object. *DataObjects* are the principle entries in the cache (discussed later).

ServiceObjects

Service objects read data from the persistent store to create data objects, and store data from data objects in the persistent store to support the Create/Read/Update/Delete (CRUD) operations and navigation operations required by the persistent object application.

ObjectExtender can generate stubs for these services if it does not understand the backend datastore.

For SQL, ObjectExtender can generate full service implementations.

The SQL statements executed are all pre-calculated and stored in methods. Currently, both dynamic and static SQL are supported.

The *ServiceObjects* have a couple of generated helper classes that understand the shape of the data objects, the shape of the SQL result rows (which may contain data for several objects) and how to map between them. These helper classes include Extractors, Injectors, and QueryPools.

For some cases, it is necessary to execute multiple statements of SQL (for example, to update, insert an object that spans multiple tables), and to extract multiple objects of differing types from a result set. This is handled transparently to the application.

For fetch-ahead, BOs are made immediately for the root object being fetched. For the fetch-ahead objects, only DOs are created and are entered in the DO cache for future use.

Modification of generated code is supported, even encouraged. This may be necessary or desirable for performance tuning and to handle complex legacy data cases.

Mapping

ObjectExtender generates service implementations for relational databases from a set of mapping specifications. The things you can specify include:

- Mapping object attributes to table columns
 - Map object attributes to one table
 - Map object attributes to multiple tables
- Mapping relationships to foreign keys
 - One-to-one
 - One-to-many
- Mapping inheritance
 - Map all subclasses to the same table with discriminator field
 - Map all subclasses to separate tables with duplicate columns for inherited attributes (requires union queries for superclass extents)
 - Map each subclass to a supplementary table containing subclass-specific attributes only (requires join queries)

Caching

ObjectExtender implements a two-level caching scheme to enhance performance.

The first level is an object cache, in the form of transaction registries that are scoped by transaction. They ensure uniqueness of BOs and help to implement the isolation required. The existence of the object cache allows the BOs to remain in the client application for the duration of the transaction without having to reread from the datastore each time they are referenced by the application.

The second level of caching is the DO cache, which comes into play during fetch-ahead. Entries in the cache are (wrappers around) *DataObjects* (DOs) and special cache entries for relationships. All DOs created from a read service invocation that are not part of the target object type's extent are placed in the corresponding DO caches until needed.

The DO cache should not be confused with the transaction registries:

- Transaction registries ensure correct object uniqueness within a transaction scope, whereas the cache is a simple optimization technique to prevent redundant access to the backend data store.
- Transaction registries are NEVER emptied until the transaction completes (although they may be weak for transactions whose isolation policy is not **Repeatable Read**), whereas the cache may be emptied arbitrarily often according to installed policies (LRU, time-expiration, and so on) without altering transaction semantics.

The DO cache is used only for pre-fetch objects. DOs are removed from the cache as soon as a BO is made for the DO. This is because the BO is being installed in a transaction registry of a transaction (and all its ancestors) and all future sharing will happen through the BOs in the registry.

There are no implemented mechanisms for flushing or controlling the size of the cache. This is a potential problem, because aggressive pre-fetch of DOs that are not subsequently converted to BOs will cause the cache to grow in an unmanaged manner.

There is no implemented policy for controlling the staleness of DO data. This is a potential problem, because arbitrarily stale data in DOs can be used for an arbitrary length of time. This problem only exists for optimistic (non-locking) transaction isolation. The following are the two scenarios that can cause problems:

- A BO gets stuck in the shared transaction for an arbitrarily long period of time without getting refreshed. This can happen so long as someone keeps a strong reference to the BO.
- A DO gets stuck in the cache for an arbitrarily long period of time. This is discussed above. These problems will be fixed in the next release, probably by allowing an age limit for DOs to be specified on a type-by-type basis. DOs in the cache that are past their sell-by date will not be used to create new BOs. BOs in transactions that do not specify **Repeatable Read** (for example, the shared transaction) whose DOs are past their sell-by date will have their DOs refreshed from the database on the next DO access.

Implementation note: The cache is a distributed structure. The cache for the DOs for a particular BO type is hung off of the *HomeCollection* for the BO. Each *Relationship* type also has its own cache. In *ObjectExtender*, related *HomeCollections* know about each other, and each network of *HomeCollection* instances include objects that represent the relationship types (these are currently instances of the generated relationship class, but this may change after the first release). If you will, the *HomeCollection* instance represents a particular BO type in the context of a data store, and these relationship objects perform the same role for the *Relationship* types. The cache for a relationship is keyed by the primary key of the BO that

owns the relationship. The relationship cache entry for a BO key consists of a collection of BO primary keys for the target BO type of the relationship. *Relationship* cache entries get filled indirectly. For example, if there is a single-valued relationship from *Faculty* to *Department*, and a *FacultyHome.allInstances* query is performed, we already have all the information needed to completely fill the cache for the *Department-to-Faculty* inverse relationship. Only queries that return complete relationship information like this one will cause the relationship cache to get loaded.

Preload

You can specify the depth to which data for an object should be retrieved from the database in a single read query.

Preload is specified in *ObjectExtender* by defining paths. An example of a path is *invoices.lineItems*. A path is relative to an existing object or relationship, and can be thought of as a sequence of relationship names to navigate through. So for a *Customer* object (or for a relationship collection of *Customer* objects), *invoices.lineItems* would be a path to all *lineItems* for that *Customer* (or those *Customers*).

ObjectExtender allows paths to be defined as part of the map for an object. Thus *invoices.lineItems* would be a reasonable path for the *Customer* object.

You can define a default preload path for a class. This does not generate extra services or protocol, but modifies the default retrieve services used by *findByPrimaryKey*: and by relationships that point to the class.

To set a default preload for a class map you could do something similar to the following:

```
| tableMap |  
tableMap :=  
  ((VapDataStoreMap mapAt: 'CeducCLIDB2')  
   mapAt: 'VapDepartment')  
   defaultPreloadPath: #('faculty').
```

This would set the *faculty* data as the retrieval depth for *VapDepartment*. Thus, whenever *VapDepartment* was retrieved from the database, *Faculty* data would be retrieved also.

Restrictions. The following restrictions apply:

- When preloading trees of objects (across relationships), pessimistic locking is supported only for the root object of the tree.
- Application control of preload paths is not supported.

Custom queries

ObjectExtender provides a simple framework that allows you to add your own custom query methods to *HomeCollections* with associated service implementations. A custom query will return a *Vector* of business objects that currently exist in the database. The query will not retrieve an object that was created by the Home but not yet committed to the database.

ObjectExtender does not support static or dynamic queries on relationship collections, nor does it automatically generate custom queries on *HomeCollections*.

DataStore

The class, *DataStore*, is responsible for owning and managing a pool of database connections. For each database connection, it registers a home collection.

ODBC restriction: The ODBC spec does not include the types BLOB and CLOB which are IBM CLI extensions to the spec. When using the JDBC-ODBC Bridge, this type is not supported. The DB2 JDBC Drivers work well with these types.

ResourceManagers

Each top-level transaction has a *ResourceManager* instance. The resource manager instance for a top-level transaction will create a *Resource* instance for each *Resource* used by the transaction.

Resources

A *Resource* instance manages a resource, for example, a database connection, for a single transaction instance.

Transaction commit flow

The flow through *Transaction* commit processing is as follows:

```
Transaction commit
  Get the ResourceManager for the Transaction and tell it
    to synchronize the transaction
  ResourceManager synchronize: aTransaction
  Sort all the versions according to the resource they belong to
    For each version in the transaction, add the version to the version list
      of the correct resource for that version
  Prepare all the resources
    For each resource sort the versions to satisfy referential integrity
  Go through the versions in order telling them to synchronize using the
    resource's session
    For each version, perform appropriate CRUD
  Commit all the resources
  Commit the DB transaction
```

External collision management

When using optimistic (non-locking) transaction isolation, ObjectExtender detects collisions on the database by overqualifying the update SQL query. You can specify which columns of the table should be included in the overqualified query in the map for the class. Use the **Be part of optimistic predicate** selection from the **Property Maps** menu item in the Map Browser.

Lite collections

A "lite" collection is a subset of an existing business object. It is an optimization feature. Creating lite collections is useful for building views that display different parts of a business object.

For example, if you were building a view to display *selective data from Course* objects for a *University*, a lite collection for *Course* would only retrieve the attributes: *name*, *credit*, and *courseNumber* from the data store.

Lite collections, therefore, read only a subset of attributes for an object. In this respect, they are very useful for displaying a choice in a list, for example. Lite collections return partially to completely populated data objects, not business objects. As a result, when displaying attributes retrieved in a lite collection, you must use the data object's methods.

Lite collections support the "drill-down" data access approach often seen in GUI intensive applications. GUI applications that open a series of nested dialogs on a set of data are prime candidates for using lite collections.

Lite collections have protocol to instantiate persistent objects. The message `#getBusinessObject` can be sent to an element of the collection to instantiate it, for example, say you have a lite collection with twenty objects and you want to instantiate the last one, you could do the following:

```
aLiteCollection last getBusinessObject.
```

The notion of "packeting" lite collections is supported as well. For example, say you have a *Student* object. Consider there are twenty thousand *Students* enrolled in the *University*. You can "packet" the amount of *Students* retrieved from the data store.

Using the *Course* scenario described earlier, a code example follows. Three lite collections for *Course* are shown:

- **byDepartment:** consists of *key*, *nameProperty*, and *dept* data, sorted by *dept*.
- **names:** consists of *key*, and *nameProperty*.
- **byCredit:** consists of *key* and *nameProperty*, sorted by *credit*.

The code would be as follows:

```
| courseClass key nameProperty credit dept |

courseClass := (Model modelNamed: 'University') className: 'Course'.
key := courseClass attributeNamed: 'number'.
nameProperty := courseClass attributeNamed: 'name'.
credit := courseClass attributeNamed: 'credit'.
dept := courseClass associationEndNamed: 'department'.

LightCollectionSpec
  name: 'byDepartment'
  namespace: courseClass
  properties: (Array with: key with: nameProperty with: dept)
  filterProperty: dept.

LightCollectionSpec
  name: 'names'
  namespace: courseClass
  properties: (Array with: key with: nameProperty).

LightCollectionSpec
  name: 'byCredit'
  namespace: courseClass
  properties: (Array with: key with: nameProperty)
  filterProperty: credit.
```

This example is just for illustration purposes. You do not need to hand code lite collections. Lite collections are much easier to create using the Model Browser.

Defining complex mappings

ObjectExtender supports defining complex mappings of object attributes to column values.

Complex mappings of a single attribute to a single column: ObjectExtender supports the notion of a **converter**.

A converter is an object that converts a column value to and from a corresponding object format. The conversion performed by a converter may be arbitrarily complex. You can code your own converter classes.

Converters are associated with columns in a schema definition, and are used wherever that schema is used. Converters encode the "real meaning" of a database column in object terms. An example of a converter is one that interprets a Y or an N in a CHAR field of a database as a *Boolean* object.

You likely will need to create your own converters from time to time. For example, you might have to convert a database integer to an IP address object. To create your own converters, subclass under *VapAbstractConverter*, which has a simple protocol that converters must implement.

Complex mappings of a single attribute to a multiple columns: ObjectExtender also supports the notion of a **composer**.

A composer is responsible for mapping a number of separate *DataObject* attribute values as a single complex *BusinessObject* attribute value.

The aggregation performed by a composer may be arbitrarily complex, and you can code your own composer classes.

Composers are associated with maps: composers define how schema values are mapped into a particular object model. An example of a composer would be one that mapped street, city and zip columns into an *Address* object.

Composers are used to create attribute values that are complex objects. These attributes do not have unique keys, and so cannot be referenced from other objects and cannot be modeled as separate top level objects with relationships to their owners.

Metadata storage model

When defining your object model, schema, and maps with the ObjectExtender browsers, in effect, you are describing metadata. The metadata is used by the framework to create the domain classes that will instantiate and service your business objects. When compared to typical class browsers, the Model, Schema, and Map browsers can be thought of as metadata browsers. That is, when browsing a class with a typical class browser, you are browsing the real class definition with all of its state and behavior. The metadata browsers are used to describe your object model to the ObjectExtender framework so that it can build a persistent layer for your business objects. Therefore, the metadata browsers are only concerned with the pertinent details to make your business objects persist. However, like the typical class browsers, ObjectExtender metadata browsers are integrated with the Envoy features so that you can save editions of your object model in the repository.

The ObjectExtender framework makes use of Envoy features to store networks of objects such as Model or Schema into a private field of a User-Defined classes. Each entity (Model, Schema or Map) is associated with and stored to a unique class. This gives you the flexibility to store and save different versions of the entity by maintaining different versions of the associated storage class.

The hierarchy of storage classes is as follows:

```

VapStorage
  ModelStorageClass
    UserDefinedModelStorageClass-1
    UserDefinedModelStorageClass-2
    UserDefinedModelStorageClass-n
  SchemaStorageClass
    UserDefinedSchemaStorageClass-1
    UserDefinedSchemaStorageClass-2
    UserDefinedSchemaStorageClass-n
  MapStorageClass
    UserDefinedMapStorageClass-1
    UserDefinedMapStorageClass-2
    UserDefinedMapStorageClass-n

```

The browsers recognize new entities that you create as well as entities that are stored in storage classes. For example, if you open a Model Browser and select **Load Available Models**, the image is scanned for subclasses of *ModelStorageClass*. For each subclass found, a model is reconstructed from the saved format. The model is then cached and displayed in the browser. This operation will not write over models which are currently cached and displayed in the browser. If you change and save the model, it will be stored back to its associated storage class. If the storage class is an open edition, then the new model, the one that you have just changed, will overwrite the previously stored copy. If the storage class was not an open edition then a new edition will be created and the new model will be saved there.

When you create a new model and save it for the first time, the Model Browser prompts you for a class and application name. If the application does not exist then it will be created.

The Model Browser also provides a way to restore the copy you are currently browsing to the match the stored copy. If you make changes and decide you want to restore the original, you can do so by choosing the **Revert model** option.

To maintain consistent sets of associated models, schemas, and maps, it is assumed that you will use these Envy features just as you would when developing applications outside the ObjectExtender framework.

For example, suppose you want to use the application named *WizBangProjectPersistenceApp* which contains the three storage classes: *WizBangModelStorage*, *WizBangSchemaStorage*, and *WizBangMapStorage* to store other versions of models, schemas, and maps.

Envy versioning could be employed to do something like this:

```

WizBangProjectPersistenceApp ProjectMilestone 1
  WizBangModelStorageClass 1.0
  WizBangSchemaStorageClass 1.0
  WizBangMapStorageClass 1.0

```

Loading this version of *WizBangProjectPersistenceApp* would make its associated storage classes (and therefore their stored metadata entities) available to the Model, Schema, and Map Browsers.

Writing your own services

If you are not using a relational database to persist your object model, you need to write your own data services. However, you can still use the code generation services to create code stubs that you can complete according to your data store requirements. After you have defined your object model, you can generate the stub service classes.

1. From the Model Browser, select your model and then select **Generate**.
2. In the Generation SmartGuide, select **Data Service Classes and Interfaces** and press **Next**.
3. Select **Stub Schema**.
4. Provide a package name where the stub service classes should be generated and press **Finish**.

Stub classes will be created for your Data Store, Data Object, and Service Object.

Defining the persistence support code

The code stubs generated by the code generation services can be completed in the appropriate manner for your specific data store. In particular, the execute methods for the service classes must be completed and the data object class must be implemented to retrieve values from the appropriate data structures.

See the implementation of the "execute..." scripts in *RelationalServiceObject* class to see how results should be cached, returned, and errors reported. These scripts illustrate the typical operations needed to work with a data store.

User code sections

In addition to providing code generation services, ObjectExtender also provides a way for you to insert your own code into methods created by these services. This open path into the generated code is provided in the event that you have special application requirements that may call for unique processing. In this way, you can take advantage of the services that ObjectExtender provides as well as add your own processing.

Using the **user code sections** in the generated methods guarantees that your code will remain intact across regenerations of your model. When you regenerate existing code, the code generator will merge the lines you added into the regenerated version of the method.

User code sections in ObjectExtender are delimited in the methods using comments.

```
methodName
    "Method comments..."

|methodTemporaryVariables |

methodCode...

"Begin user code {section name}"
"End user code"
```

User code sections are generated in the following classes:

- *BusinessObject* attribute primitive getter methods (pre/post-Get)
- *BusinessObject* attribute primitive setter methods (post-Set).
- *BusinessObject* attribute public getter methods (post-Get).
- *BusinessObject* attribute public setter methods (pre-Set).
- *BusinessObject* relationship public setter methods (pre/post-Set).
- *BusinessObject* *preStore*: (pre-Store).
- *Key* *initializeFromBO*: (post-initialize).
- *Key* *setInBO*: (post-Set).
- *DataObject* attribute setter (pre-Set).
- *DataObject* attribute getter (pre-Get).

User code sections are generated in the business object, key, and data object classes.

In some cases, specific methods are noted. In other cases, the generic getter/setter terminology is used indicating attribute accessor methods. The names of the user code sections are given in parentheses. These names are found in the comments of the user code section. Think of them as labels for the user code that suggest what is to be done at a particular linear section in the code.

```
accountNumber
  "Get the value of the attribute accountNumber in the current transaction"

  | value |

value := self bom versionDataForRead primAccountNumber.

"Begin user code {post-Get}"
"End user code"
```

Managing business objects

Once you have created your model, schema, and data store map, and generated their supporting code, you can then create the application layer that will manage the business objects. If you plan to create a user interface, you can use the parts in the *ObjectExtender* palette category on the Composition Editor. These parts are transaction-aware, that is, they understand transaction semantics such as *#begin*, and *#commit*.

Depending on your requirements, you can build nearly all of your application using VisualAge tools with very little scripting; you can build the persistence layer with the *ObjectExtender* tool set, and the user interface with the Composition Editor utilizing the *ObjectExtender* parts as well as the other parts available for building user interfaces.

If your application will not utilize a user interface, there are some details you will need to code in the code to manage your business objects, such as activating the data store, beginning transactions, creating business objects, and so on.

This section collects some of the details useful for programmers who are applying business rules and application logic to the model domain.

Using the data store

Activating the data store. Before attempting to read or write any *business objects*, you must activate the appropriate data store. This is done by sending `#activate` to the singleton of your data store class. This will associate the appropriate service classes to the home collections, and initialize any required database connections. A sample data store activation would be similar to the following:

```
yourBusObjDataStore singleton activate
```

When testing your code, you can also activate a data store using the Status Tool.

Creating, retrieving, deleting instances

Creating business objects. There are a number of ways to create persistence instances. The first is to create a persistent instance with the create protocol on the home collection.

Before this can be done, a data store and non-read-only transaction must have been activated. There is one data store for each map that has had services generated from it and the map provides the layer between the model domain objects and the persistent back end.

Having created an object with the create protocol, it can be modified and/or deleted in the transaction in which it was created or within a transaction that is nested from the transaction in which it was created. If any attempt is made to access the object in a transaction other than these, a `VapVersionNotFound` exception will be raised.

Before persisting the object (which will happen when the `TopLevelTransaction` is committed) the key must have been supplied to the object. The key is the set of attribute and association roles that were specified as the object identifier (OID) in the Model Browser. For example, if the object has a key of name, the following protocol would be sufficient:

Another way of creating an object with a specific key is with the `createFromKey` method on the home collection:

The code generation services will create a helper method that has a specific create method that has one parameter for each OID element.

Retrieving business objects. Instances of business objects can be retrieved by creating an instance of the appropriate key class and asking a home collection to find the corresponding instance.

```
DepartmentHome singleton findByPrimaryKey: (DepartmentKey with: 'CIS')
```

Asking for all instances. All instances of a persistent class can be retrieved by sending a message to the home collection for the business object. This will always execute a query against the database to retrieve a Vector of business object instances. This method returns all instances that exist in the database, not the instances that exist only in the image.

Deleting business objects. Send the `remove()` message to primary objects to delete them, and `secondaryRemove()` for deleting cascaded objects.

Accessing relationships of a business object

You can access the relationships of a business object instance by invoking the generated relationship accessors. When the relationship is many-valued, the result will be a specialized collection, which provides transactional support for relationship collections. It responds to most collection protocol, and will return the correct relationship value for the current transaction.

Coding transactions manually and visually

Creating transactions. You can begin a new `TopLevelTransaction` with:

```
Transaction begin: 'MyTransaction'.
```

Creating nested transactions. You can begin a new nested transaction by sending `#beginChild:` to an existing transaction:

```
aTransaction beginChild: 'MyChildTransaction'
```

Creating named transactions. Transactions can also have names to identify them in debugging and to aid in collision management. Refer to “Transaction overview” on page 91 for more details.

Commit and rollback strategies. Transactions can be ended with a variety of commit and rollback methods which allow the specification of success and failure actions. Refer to “Transaction API” on page 96 for more details.

Transaction isolation policies. Transaction isolation policies can be specified for each Transaction. These policies control the behavior of transaction view version copying, merging, and promotion to the shared view.

ObjectExtender Parts

Several parts are available to enable you to work with transactions visually.

SharedTransaction. This part represents a distinguished or singleton instance of the `SharedTransaction`.

TopLevelTransaction. The `TopLevelTransaction` part represents a top level transaction. When this part is used in a composition editor it will create the top level transaction during the initialization phase of the view. This helps to ensure that the view’s current transaction has been switched to the top level transaction before any of the connections are initialized.

TransactedVariable. In some circumstances, the applet or view is only working with a single current transaction. In this case the standard variable part (supplied in `VisualAge`) can be used to allow objects to be connected together and to the user interface parts. In some views, however, there may be several active transactions and the programmer needs to ensure that a specific variable on the view is “pegged” to a given transaction. This is done with the `TransactedVariable` part. The transacted variable has all of the behavior of the standard variable part except that it has one additional property: `transaction`. When the transacted variable part is given a transaction it will ensure that all connections are done with that transaction as the current transaction. Having executed the connection, the previous current transaction will be resumed. This allows connections to be guaranteed to get and

set the value of the object within the transacted variable that is for the transaction that has been specified, irrespective of what the current transaction is at the point when the connection is activated. With the standard VisualAge variable part, the current transaction would be the one in which the get or set method was executed.

BusinessTransaction In some view scenarios, when a transaction is committed, you may want the view to remain open, in other scenarios, you may want to close the view. For example, an OK button might commit a view's changes and close the view whereas a Save button might commit the changes and keep the view open.

Whenever a transaction is committed, its state changes from active to committed and its parent transaction will be resumed. Any attempt to resume the committed (or rolled back) transaction will cause a `VapTransactionFailure` exception to be thrown. The developer must therefore begin a new transaction to allow the user to remain in the view at the same transaction nesting level.

The business transaction caters for such a scenario. It has a read only property of transaction which it will lazily initialize to a new `TopLevelTransaction` (Lazy initialization means that the transaction is created when it is first asked for if it has not been previously created and is null). When its transaction is committed (either through the `commit()` and `rollback()` methods on the `BusinessTransaction` part or by the transaction being committed elsewhere) a fresh transaction will be created by the part. When the transaction is regenerated the part signals the transaction property (which is bound to the transaction event). If the transaction property of the `BusinessTransaction` is connected to the transaction property of a `TransactedVariable` this will cause the `TransactedVariable` to refresh itself with the version of its object in the new transaction and the user is allowed to continue working with the transacted variable's contents.

By default the `BusinessTransaction` will create a new `TopLevelTransaction` the first time its transaction property is retrieved. Rather than create this as a child of the `SharedTransaction` the part will create a private `ReadOnly` transaction that it uses as the parent of its transaction. The reason for this is that when a transaction is committed its parent will be resumed. If a `TopLevelTransaction` is committed the `SharedTransaction` will be resumed. The `SharedTransaction` is not able to have any new objects created into it, including read only objects that are newly read in from the data store. By having the `ReadOnly` transaction as the parent however ensures that in the gap between the transaction being committed and refreshed the current transaction is able to read and refresh objects. This behavior can be toggled with the property `createReadOnlyParent` on the part.

It is also possible to explicitly set the parent transaction that the `BusinessTransaction` part should use to create its child transaction. This is done through the writable property `parentTransaction` on the part. This allows views that have to support nested transactions to be constructed by connecting the transaction property of one `BusinessTransaction` to the `parentTransaction` property of another. The latter transaction will be a nested child of the former, and the `BusinessTransaction` parts will ensure that whatever combination of transactions get committed or rolled back that both parts have valid transactions that are nested from one another.

Because the business transaction lazily initializes its transaction when it is first asked for, there are some circumstances where this is done before the parent transaction is set. In this case it may be the desired behavior that the previously lazily initialized transaction (which will not be a child transaction of the new parent transaction) should be rolled back. This is the default behavior. If this is not

the desired behavior, that is, when the parent transaction is set on the part any previously generated transactions are left active, the boolean property `rollbackTransactionWhenParentChanges` should be toggled.

Analyzing performance

There are different ways you can analyze and tune performance for persistence. The Status Tool can be used to monitor the transactions, views, and caches in the system. You can use it to reset the state of various components as you unit test and get your code working.

Executing SQL. The SQL Query Tool can be used to execute SQL code against the data store connection.

Tracing code paths. The *Trace* class provides a singleton trace object that has a default output stream and switches for trace levels.

Debugger settings: Catching exceptions. You can also take advantage of the **Workbench-Window-Debugger-Caught Exceptions** dialog to get better debugging for exceptions thrown within try/catch or synchronized blocks.

Appendix. Restrictions

In addition to restrictions, this section documents workarounds and suggestions that may be useful to product users.

Home createFromKey:

When creating a new object, you can create it with the Home createFromKey: method. If that key contains relationships (really the keys of the related objects) then the related objects cannot be new objects. If you want to set a relationship to a new object, you should use create and then set the relationships with the new object (NOT the key), or use the createWith:with: helpers on the homes. The reason behind these restrictions is that the key of a new object is not guaranteed to be permanent, so we do not support lookup of a new instance by key, therefore a relationship to that new object by key alone can't be resolved. For example:

```
| b1 newBranch c bc |  
b1 := VapBankBranchHome singleton findByBranchNumber: 'BR001'.  
newBranch := VapBankBranchHome singleton createWithBranchNumber: 'BR999'.  
c := VapCurrencyHome findByCurrencyType: 'CRC01'.
```

```
"Relate two existing objects: Works OK"  
bc := VapBranchCurrencyHome createForBranch: b1 currency: c.
```

```
"Relate one existing object and one new using helper:  
Sets relationshipvalue directly, Works OK"  
bc := VapBranchCurrencyHome createForBranch: newBranch currency: c.
```

```
"Relate one existing object and one new using createFromKey:  
Does not work!"  
bc := VapBranchCurrencyHome  
createFromKey: (VapBankBranchKey with: newBranch key with: c key).
```

```
"also: until newBranch is committed to the data store and the  
Shared Transaction, you can't find it by key,  
ala: VapBankBranchHome singleton findByBranchNumber: 'BR999'. "
```

Residual instance variables on model regeneration

If you delete class attributes in a model for which you have already generated code, and then regenerate the model, you may encounter the following problem: your instance variables represented by the attributes you deleted will still exist in the class, however their getter and setter methods will be properly deleted.

DBCS character text field entry

You cannot enter DBCS characters in ObjectExtender browsers and tools except for one case:

1. **Physical** name for tables, columns, and keys.

In addition to **Physical** name, there is a corresponding (optional) logical name field (**Name**) for tables. If you import a schema from a database, which has double-byte characters, enter logical names for each physical name, you must enter single-byte characters for the logical name.

Mapping, relationships, and preload limitations

Inheritance Mappings

1. Disinheritance not supported.
2. Remapping of superclass map ivar/relationship maps not supported.
3. Mixing inheritance strategies within one mapped inheritance hierarchy is untested.
4. Use of discriminators outside of inheritance mappings is untested.
5. Multiple column discriminators is not supported.
6. Discriminators that are also keys is untested.
7. Relationships among classes within a mapped inheritance hierarchy has not been fully tested.
8. Distinct table inheritance is not supported.
9. Use of foreign key relationships that are not "primary key to primary key" in root/leaf inheritance mapping is untested

Abstract Mapped Classes

1. Abstract mapped classes are supported only within inheritance mappings. We assume that the abstract class is mapped to an (abstract) table in the database. Otherwise abstract mapped classes are not supported.

Relationships

1. Relationships across data stores are not (fully) supported
2. Relationships involving "non-standard" foreign keys are not supported, for example:
 - a. keys that do not match either with the number columns or column types
 - b. keys involving transformations of the key data
 - c. keys involving column or database functions
3. Many to Many relationships with hidden tables are not fully supported

Class Mapped to Multiple Tables

1. Foreign key relationships that not "primary key to primary key" (i.e. "backward" 1 to 1) are not tested
2. Classes split over data stores are not directly supported

Preloading. There is a known bug with preloading relationships on a class whose target classes are mapped to the same class (or inheritance hierarchy). The query generated is incorrect. This occurs in the following scenarios:

1. Class with relationship to self (or some other member of its own inheritance hierarchy).
2. Class with two or more relationships to the same target class (or classes in the same inheritance hierarchy).
3. Relationships between classes in the same inheritance hierarchy.

Single character data and vendor DB drivers

Various vendors' database drivers treat single-character data differently. Some will return a Character, some will return a String of length one. To be safe, if an application may run with multiple drivers, type Char(1) fields as strings, and the VapCharToString converter will guarantee that a string will always be returned.

IC instruction not preloaded

For all VisualAge users using ICs, they must load the following two config maps before using ICs, 1) Envy/Packager IC Instructions, and 2) VisualAge IC Instructions

For ObjectExtender users, you must load one more instruction: VisualAge Persistence IC Instructions

These instructions must be loaded in this order.

Composed object does not make aggregate dirty

When using composed objects, the aggregate object is not marked "dirty" in a partial update scenario. For example, TstCustomer object has a name which is a composed object. When an update partial name operation is performed (first name or last name) the object is not marked dirty, so the changed object will not be written into the data store. To force the change, you must change the other attribute of the object.

Global reset does not clean up database connections

If you attempt to clean up database connections using the "Global ObjectExtender Reset", and you get an unknown abtError, you may still have a dead connection to the DB. One workaround for this is to evaluate the following:

```
AbtDbmSystem startUp
```

Generate models first before other code-gens

As a general rule, always generate code for models before generating services code or stub code for schema or local image persistence. If you generate code without a model present a walkback will occur stating that the model is missing.

Backwards 1:1 relationships

If you have backwards 1:1 relationships, you must always regenerate the services after generating the model. Model generation assumes that all 1:1 relationships are forwards, service generation uses the maps to correct those which were backwards.

When Changing composer types

If you have a complex attribute map, and you want to change the Composer type, you need to delete the Complex attribute map from the Map Browser and recreate it in the Property Map Editor.

Specialization for cascade delete

A specialization of BusinessObject>>#abtRemove is available for cascade delete operations. The cascade responsibility is divided between the two remove methods. #abtRemove is sent by the application to the root object being deleted. This method can be overridden to disconnect the root object from its parent object. The default implementation of #abtRemove just calls #markRemoved. #markRemoved should be specialized to perform any disconnects and cascades which an object should perform whether or not it is the root. It should not disconnect from its parent object here. It should not disconnect any connections to child objects, since these connections are required to perform operation sorting for referential integrity constraints. An example might go like this:

```

VapCustomer>>#abtRemove
  self bankBranch: nil. "disconnect from parent"
  super #abtRemove "calls #markRemoved"

VapCustomer>>#markRemoved
  self accounts do: [:acc | acc markRemoved]. "cascades to transactions, etc."
  self homeAddress markRemoved.
  self billingAddress markRemoved.
  "you could also disconnect any non-aggregate associations here"

```

Foreign keys used in 1-1 relationships

If foreign keys are used for 1-1 relationships, the association must be navigable. For example, suppose you have the following :

```

Model:
  Customer <-----> Address
    1. homeAddress
    2. billingAddress
Schema:

  Customer table contains homeAddress and billingAddress columns
  which are foreign keys to Address table.

```

You must then define both model associations navigable. Otherwise, on inserting customer, it will not insert the address first, and you will get an insert customer error because the address parent key does not exist.

Service Generation should generate #executeSingleUpdate: for No key object

The problem occurs when a model class has only properties which are part of its OID.

An association object, modeling a join table, is a good example.

Say you have a model class called BranchToCurrency which is an association between Branch and Currency, its only properties are two associations, and these two associations make up the OID of the BranchToCurrency object. Since the database row only has key fields, there are no updatable fields, and our UPDATE query will not work properly.

The workaround is to override the service method #executeSingleUpdate: to do nothing. The application developer should structure their use of these kinds of objects so that the application never tries to update one, always adding or removing new instances rather than changing an existing one. The override above is just to catch a case where the app accidentally updates an object, maybe by disconnecting a related object.

Pre-req for user-defined composers

Model and service applications which use Composers need to make sure their pre-reqs are updated to include their application which define any user-defined model and composers. For example, if a model has a property which is a Name, and the maps specify that a composer called NameComposer should be used, then the generated model application must pre-req the application defining Name, and the generated service app must pre-req the application defining NameComposer.

No converter for AbtMonetaryAmount

A converter for AbtMonetaryAmount is not provided, use ScaledDecimal instead.

Workaround for Sun core exiting VisualAge ObjectExtender application

On Soliaris platforms, if you encounter a core dump on exiting VisualAge ObjectExtender application (runtime), the workaround is to comment the method #shutDown in class PlatformLibrary, then re-package the application.

Glossary

ACID. A mnemonic for the properties a transaction should have to satisfy the Object Management Group Transaction Service specifications. A transaction should be Atomic, its result should be Consistent, Isolated (independent of other transactions) and Durable (its effect should be permanent).

atomic. An atomic database transaction is one which is guaranteed to complete successfully or not at all. If an error prevents a partially-performed transaction from proceeding to completion, it must be "backed-out" to prevent the database from being left in an inconsistent state.

BO. Business object. This term denotes the objects in your problem domain that you wish to persist in a data store.

cardinality. Cardinality expresses the constraints on the number of instances that are related through a relationship.

code-generation. The function whereby code is generated automatically given certain specifications.

CB. Component Broker.

connect/disconnect. The attachment (or unattachment) of a target business object to/from a link.

consistency. In the context of ACID: a transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program.

counter. Referring to the other side of a two-way business object relationship.

CRUD. Create/Read/Update/Delete, the four basic types of operations on database rows, records. ObjectExtender provides these operations on objects as well.

data model. The term, data model is used here, to make a distinction between it and an object model that you wish to persist. For example, the schema for a relational database represents the data model. Your object model is represented differently and will not need to be tightly coupled with the data model representation.

DDL. Data Definition Language. A language enabling the structure and instances of a database to be defined in a human- and machine-readable form.

DO. DO is an acronym for data object. Data objects contain the data for the business objects. The data is in the form in which it was retrieved from the data store.

durability. In the context of ACID: Once a transaction completes successfully (commits), its changes to the state survive failures.

framework. In object-oriented systems, a set of classes that embodies an abstract design for solutions to a number of related problems.

home collection. Home collections provide the logical home for business objects. They provide APIs for creating or locating instances.

hydration. The activity of populating the properties and relationships of a model object.

isolation. In the context of ACID: even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.

link. The infrastructure that connects source and target business objects in a relationship.

metadata. Any information which describes according to prescribed specification a target data.

multiplicity. See cardinality.

nested transaction. A nested transaction is a tree of transactions, the sub-trees of which are either nested or flat transactions. Transactions at the leaf level are flat transactions. The transaction at the root of the tree is called the top-level transaction; the others are called subtransactions. A transaction's predecessor in the tree is called a parent; a subtransaction at the next lower level is also called a child. A subtransaction can either commit or roll back; its commit will not take effect though, unless the parent transaction commits. Therefore, any subtransaction can finally commit only if the top-level transaction commits. The rollback of a transaction anywhere in the tree causes all its subtransactions to roll back.

object model. Object model is to data model what a hierarchy of classes is to a schema of database tables. It is simply a distinction between the two representations of how the data is represented.

OO. Object-oriented. Can apply to analysis, design, and programming disciplines.

persistence. A property of a programming language where created objects and variables continue to exist and retain their values between runs of the program.

This is in contrast to transient objects that cease existing when the application that created them is not running.

pre-fetch. The notion of defining a path to set of data that you want to preload from data store to object model to reduce the number of database trips improving performance.

relationship. As understood in the context of the ObjectExtender framework, a relationship is an instance variable in a business object which contains a reference to another persistent object.

transaction. A unit of interaction with a DBMS or similar system. It must be treated in a coherent and reliable way independent of other transactions.

UML. Universal Modeling Language.

Index

B

BusinessObject
 accessing 113

C

caching 3, 103
cardinality 13, 32
code, user 110
code generation options 29
collision management 94
 examples 97
 Repeatable Read 94
 locking implementation of 95
 non-locking implementation of 95
 Unrepeatable Read 94
 locking implementation of 96
 non-locking implementation of 95
commit
 transaction flow 106
complex mappings 107
 multiple attribute to single
 column 108
 single attribute to single column 107
composer 40
converter
 adding your own 108

D

DataObject 103
DataStore 6, 106
 activating the 112
DBCS 117
debugging 115
 gather statistics 115
 monitoring 115
 query 115
 Trace 115

E

entry path 19
Envy
 managing versions of models 108
event signalling 101

external collision management 106

F

fetch-ahead 3, 103, 105
 restrictions 105, 117
 setting a default 105

G

generating
 schema to model 30

H

HomeCollections 90

L

lite collections 31
loose coupling 2

M

mapping 14, 103
 attributes 103
 inheritance 103
 relationships 103
metadata storage model 108
minimal intrusion 2
multiplicity 13, 32

O

object uniqueness 99
 registry 99
 weak structure 100
optimization 2, 3, 103, 105
 restrictions 105, 117
 setting a default 105

P

paths of entry 19
prefetch 3, 103, 105

prefetch 3, 103, 105 (*continued*)
 restrictions 105, 117
 setting a default 105
preload 3, 103, 105
 restrictions 105, 117
 setting a default 105
programming model 89

Q

queries 5, 105
query 5, 105

R

relationship 12
 cardinality 13
 inverse 13
 ownership 13
Resource 106
ResourceManager 106

S

service object 103
shell business object 100

T

Transaction 94
 API 96
 concurrent 4
 nested 4
 registries 104
 shared 99
 top-level 99
transaction isolation policy 94
 examples 97
 Repeatable Read 94
 locking implementation of 95
 non-locking implementation of 95
 Unrepeatable Read 94
 locking implementation of 96
 non-locking implementation of 95

U

user code section 110