Cincom®

Cincom Smalltalk™ VisualWorks®

Glorp Developer's Guide

VisualWorks 8.3

P46-0151-03

SIMPLIFICATION THROUGH INNOVATION

Notice

Part Number: P46-0151-03

Software Release: 8.3

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

Preface

GLORP (Generic Lightweight Object-Relational Persistence) is an open-source object-relational mapping framework for Smalltalk. Glorp is implemented for several Smalltalk dialects.

For VisualWorks, Glorp is increasingly being used for critical functionality, specifically in the Store source code repository management system. VisualWorks also gives full access to Glorp functionality, making it available for use by all VisualWorks users.

This document is made available to fill a gap in the Glorp documentation, providing essential architectural information and usage instruction.

Although the Smalltalk dialect used throughout this document is VisualWorks, the information and examples should apply to other dialects as well, with minimal and routine modifications.

For additional information about Glorp, refer to the project web site.

Audience

The discussion in this book presupposes that you have at least a moderate familiarity with object-oriented concepts and the VisualWorks environment. It also presupposes that you have a basic understanding of database applications.

For introductory-level documentation, you may begin with a set of on-line VisualWorks Tutorials, and for an overview of Smalltalk, the VisualWorks development environment and its application architecture, see the *VisualWorks Application Developer's Guide*.

Conventions

We follow a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Examples	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`c:\windows`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
filename.xwd	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks tools.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File > **Open...**	Indicates the name of an item (**Open...**) on a menu (**File**).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	Select (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks window (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	**2-Button**	**1-Button**
<Select>	Left Button	Left Button	Button
<Operate>	Right Button	Right Button	<Option>+<Select>
<Window>	Middle Button	<Ctrl> + <Select>	<Command>+<Select>

Additional Sources of Information

Cincom provides a variety of tutorials, specifically for VisualWorks.

The guide you are reading is but one manual in the VisualWorks documentation library.

Complete documentation is available in the `/doc` directory of your VisualWorks installation.

Chapter

1

Introducing Glorp

Topics

Glorp provides a complete facility for storing your application data in commercial or open-source relational databases such as Oracle or PostgreSQL. It is an open source, platform-independent framework that works on all popular dialects of Smalltalk. To simplify the design of your application, the framework manages all mapping between relational and object data models, and enables true object-level transactions.

For applications with simple domain models, the Active Record API alone is often sufficient. This API has been implemented as a layer on top of Glorp, such that if your application requires more complicated modelling or mapping features, the lower-level API is always readily available.

Aside from vendor-specific database libraries, Glorp is a self-contained framework, providing all the Smalltalk code interfaces required to persist your application data. As a developer, you do not need to learn or invoke any other database framework (e.g., the VisualWorks EXDI).

Overview

This guide provides an overview of object-relational mapping, including discussions of key concepts. Step-by-step instructions for designing and building your data model, as well as detailed discussions of the interfaces and code patterns used to persist your application data. While this guide has been written for VisualWorks application developers, the Glorp APIs are the same on all dialects of Smalltalk.

What is Glorp?

Glorp is an acronym for Generalized Lightweight Object-Relational Persistence.

Glorp provides the following features:

- Scalable support for both simple and complex domain models
- Declarative mapping via metadata
- Complex and dynamic queries
- Transparent, non-intrusive management of relationships
- Partial reads and writes of objects
- Write barriers
- Object-level transactions
- Object-level queries
- Reads and writes can be optimized for performance

Supported Databases

Supported databases include: Oracle, Oracle ODBC, PostgreSQL, DB2, SQLite, MySQL, Ocelot, Adabas, InterBase/Firebird, Microsoft SQL Server, and Microsoft Access.

Note: Some of these may not be fully supported for all features of the Glorp framework, in particular MySQL and MS-Access. If the VisualWorks EXDI implementation for a database is an unsupported goodie, the same status applies to Glorp's use of it.

For information about the EXDI connection to specific databases underneath Glorp, consult the Database Application Developers Guide.

Loading Glorp

All support parcels for Glorp may be found in the `/glorp` subdirectory of the VisualWorks distribution. In addition, the `/preview` subdirectory may contain add-on utilities.

To load the database support parcels into your VisualWorks image, open the Parcel Manager (select **System > Parcel Manager** in the Launcher window), select **Database** under **Suggestions**, and load the Glorp parcels by double-clicking on the desired items in the upper-right-hand list of the Parcel Manager. You must also load the EXDI parcels for whichever database(s) you wish to use.

For convenience, most of the code examples in this document are available in the GlorpGuideExamples parcel. Other examples can be found in GlorpTest parcel.

Object-Relational Mapping

Object-Relational mapping is a technique for translating data between object-oriented and relational systems, used to persist domain models implemented as Smalltalk objects. Due to the relative complexity of each system and the "impedance mismatch" between them, object-relational mapping is generally provided by a special framework for application developers who wish to persist objects in a commercial database, or to manipulate relational data.

For simple applications, the Active Record pattern may be used. It works by requiring a specific pattern to relate domain objects and tables. Thus, it is unsuited to complex or legacy cases. Glorp supports the Active Record pattern, but is a much more general framework.

While is it possible to perform some simple mappings between tables and instances of classes in an object-oriented language, the underlying paradigms of the two technologies ultimately do not allow a seamless translation between them.

For example, while SQL types such as `String`, `Number`, etc. can be mapped to corresponding Smalltalk classes, care must be exercised when mapping to types that do not possess identical dimensions. Complex data types pose a further difficulty. Where Smalltalk provides a rich set of collection classes such as `Dictionary`, `Bag`, and `SortedCollection`, by contrast RDBMS provides a more limited repertoire of tables and blobs. Moreover, objects and tables do not observe the same rules of internal consistency.

Generally speaking, where relational systems use a declarative approach, object-based systems generally use a behavioral approach. Objects do not follow an explicit schema, instance variables in a class are untyped or may be polymorphic. In contrast, tables in a relational database are typically defined using schemas. Similarly, classes inherit behavior from superclasses, but no abstraction corresponding to inheritance exists for relational tables. The former tend to find expression as behavior trees, while the latter are set-based taxonomies.

Glorp provides a way to bridge between the semantics of relational and object-based systems. It handles mapping and provides object-level transactions. By using this framework, you do not need to embed SQL statements into your application or domain code. SQL is automatically generated by the famework, but rather than providing an API to do this, *metadata* is used.

While you can develop an application and use Glorp without a detailed understanding of SQL, you do need some knowledge of SQL concepts to use it effectively. Yet, instead of thinking about how SQL is being generated inside the Glorp framework, developers only need to concern themselves with metadata that describes the domain model. This results in a simpler, smoother, and potentially more efficient way to persist your application data.

Metadata

Since Smalltalk represents data using classes rather than types, there is of course no DDL (Data Definition Language) such as one finds in SQL. Object-Relational mapping requires not only a mechanism to translate between objects and tables, but something to express their correspondence. To achieve this, Glorp uses special declarative descriptions generally referred to as *metadata*. The structure of this metadata is one of the basic concepts that developers need to understand.

This metadata is used when handling reads, writes, joins, and anything else that happens during database transactions. Most of this is transparent and handled automatically. In this way, developers do not need to write SQL code, because the metadata handles the mapping between Smalltalk objects and their database representation. Behind the scenes, this metadata facilitates the automatic generation of appropriate SQL expressions, so your application doesn't need to.

The metadata is written in Smalltalk, not SQL, and is included as part of your application. I.e., it is the developer's responsibility to write the metadata. You may simplify this task by using Glorp’s Active Record framework, or the Atlas utilities in VisualWorks, or the (powerful) Glorp Atlas Mapping Tool in ObjectStudio.

Mapping

Most object-oriented persistence frameworks use classes to model database tables, the basic idea being to represent each domain class using a separate table. The class' instance variables are represented using columns in the table, and each instance of the class corresponds to a row. Glorp, too, follows this general pattern.

As part of the metadata associated with the domain model, Glorp includes *mappings* that define how domain classes are mapped into and out of the database. Several flavors of mapping are available, each defining a specific way to translate between a Smalltalk object and a column in a relational table. Typically, each instance variable in a domain class is handled with a distinct mapping object, and these are all gathered together in a single class known as a *descriptor system*.

By expressing object-relational mappings as metadata, your application does not need to concern itself with mapping between Table, Row, or Column objects, and your domain objects, as in some older persistence frameworks. Similarly, Glorp does not require code in each of your domain classes to read and write objects.

For each application, then, Glorp presupposes a single domain model whose meta-representation is expressed using a single descriptor system class. The domain model may be composed of many distinct classes, but there is only one descriptor system class.

The task of adding metadata and mappings for your domain model involves creating this descriptor system class, either wholly by hand, or with the assistance of Active Record or the Atlas Utilities.

Objects, Transactions, and Units of Work

Once your domain model has been annotated with metadata in a descriptor system class, your application can store and retrieve domain objects from the database. To do this, your application uses a `DatabaseAccessor` to establish a connection to the database, and a session object to coordinate queries, reads, writes, etc.

As a rule, the content of a database is always manipulated within the scope of a *transaction*, in order to ensure data integrity. However, since Glorp works with objects, rather than rows from tables, the requirements for transaction processing are slightly different. Instead of transactions, your application should use part of the session API to execute *units of work*. A unit of work is similar to a transaction, except that it automates the manipulation of complex objects that may span several tables, and it can roll back changes to Smalltalk objects if the transaction is cancelled.

To provide this behavior, Glorp includes an *object cache* that maintains object identity. Any objects that are manipulated during a unit of work enter the cache, which may be used to restore objects to their previous state in the event of a rollback. In effect, a unit of work is an object-level transaction. The API for initiating, committing, and rolling back units of work functions at a higher level than the basic transaction API for the database server. Thus, while it is possible to use Glorp as a glorified communication channel to the database server (i.e., like the VisualWorks EXDI), or to use classes as storage managers, the framework is intended to be used with more powerful abstractions such as mappings, descriptors, and units of work. These, then, are the basic concepts you need to understand when using Glorp.

Utilities

Glorp includes several utilities that can help when building the descriptor model for your application:

Glorp Atlas Utilities

For details, see the "Atlas" chapter in this guide, and comments in the `GlorpAtlasUI` parcel and its prerequisites.

Glorp Atlas Mapping Tool

A powerful tool in ObjectStudio. For details, see the *Mapping Tool User's Guide* in the ObjectStudio distribution.

Chapter

2 Database Control

Topics

The Glorp framework provides a control layer for interacting with a database server via client libraries located on your workstation.

The database server may be running either as a service on your workstation, or it may be located on another machine. In either case, this server is responsible for managing all database storage files, and performs database operations on behalf of each client. To use the database, a client opens a connection with the server, and issues commands using a session object. Within each session, units of work ("transactions" in the parlance of relational databases) may be executed.

Managing a Database Connection

A database connection is established using a DatabaseAccessor object and a Login object.

A DatabaseAccessor provides a vendor- and dialect-independent mechanism for connecting to and disconnecting from a database.

A Login object holds connection-specific parameters such as the username, password, and environment string.

Creating a Login Object

A Login object holds all the information required to connect and disconnect from a database. In fact, the connection is managed by a DatabaseAccessor, which requires a Login object as a parameter.

A Login object can be created once at configure time for each database, saved in a shared variable belonging to the application model class, and used when subsequently establishing connections to the database.

For maximum security, your application should set the password each time the Login object is used to establish a connection.

In addition to the familiar connection parameters (e.g., username, password, and connection string), a Login object requires a DatabasePlatform object to specify the particular vendor-specific characteristics of the database. For all supported databases, your application need only provide the name of the appropriate subclass of DatabasePlatform, using the database: message.

For example, to create a Login object for a Postgres database:

```
aLogin := Login new database: PostgreSQLPlatform new;
 username: 'username';
 password: 'password';
 connectString: 'host' , '_' , 'db'.
```

By default, Login objects immediately forget their password after a successful login. To manage this behavior, each Login object includes an instance variable named `secure`, which is set to `true` by default. If you wish to change this, use the `secure:` message, as follows:

```
aLogin secure: false.
```

Creating a DatabaseAccessor

A Glorp `DatabaseAccessor` provides an abstract, vendor-neutal interface for connecting to databases and issuing commands. It also encapsulates all dialect-specific code used to access databases in various flavors of Smalltalk.

Instances of `DatabaseAccessor` (actually, one of its subclasses) are used as connection objects. Once you have created a `Login` object, you create a platform-specific `DatabaseAccessor` subclass instance like this:

```
accessor := DatabaseAccessor forLogin: login.
```

To actually issue commands through a database accessor, a session object is used. As a rule, each accessor object belongs to only one session object.

If you are building a web application that needs connection pooling, the pattern for creating an accessor is slightly different. For details, see the (separate) discussion of connection pooling.

Connecting to a Database

To connect to a database using a specified accessor:

```
accessor login.
```

To disconnect from a database using a specified accessor:

```
accessor logout.
```

Testing a Connection

To test a database connection, use a `DatabaseAccessor` object:

```
[accessor
 login;
```

```
  logout]
   on: Error
   do: [:ex | Transcript show: ex messageText].
```

If the accessor is unable to access the database (e.g., the Login object has incorrect parameters), an exception is raised.

Using a Connection Pool

To better manage database resources and improve performance, web applications should generally use connection pooling. This feature is available for Glorp as part of the Seaside-Glorp package.

Support for pooling is provided by class VWPoolingDatabaseAccessor, which you should use in place of DatabaseAccessor. E.g.:

```
poolingAccessor := VWPoolingDatabaseAccessor forLogin: aLogin.
poolingAccessor login.
```

The accessor object is intended to be maintained as application state, and used when new sessions are created. In this case, rather than using sessionForLogin: to create a session, you should build the session object like this:

```
aSession := (GlorpSession new)
    accessor: poolingAccessor;
    system: MyDescriptorSystem;
    yourself.
```

The pool of connections is automatically maintained by the accessor object. When your application performs a unit of work, the session object takes a connection from the pool, executes a transaction, and then returns the connection to the pool. All of this happens transparently and requires no intervention on the part of your application.

Executing SQL

Using a DatabaseAccessor object, you can directly execute SQL on a database. This is generally discouraged and does not make use of the full power of Glorp, but there may be occasions when you may need to do this.

To execute a String containing a SQL expression:

```
accessor beginTransaction.
result := accessor executeSQLString: aSQLString.
accessor commitTransaction.
```

The result object is an Array of values representing the rows returned, if any.

If the string passed to executeSQLString: contains invalid SQL, an exception will be raised. To catch any exceptions that occur, use an exception handler, e.g.:

```
[result := accessor executeSQLString: aSQLString]
 on: Error
 do: errorBlock
```

Again, whenever possible, you should consider using Glorp's higher-level APIs.

Creating a Session

A *session* coordinates most of the interaction between your application and the database. It acts as a broker for queries, manages the object cache, the descriptor systems, and units of work. A session also provides a mechanism for registering modified objects. Each session is represented using a distinct instance of GlorpSession.

Each session requires a database connection, and generally each connection supports a single session. Normally, your application provides a distinct Glorp session per user, and uses connection pooling to manage connections in multi-user (e.g., web) applications. Apart from being used for one connection, each Glorp session maintains a set of objects, and should only be used for one concurrent unit of work or transaction.

When a unit of work is completed, the session object decides whether objects need to be written to the database, whether the rows need to be inserted, updated, or deleted. The session object is thus also responsible for generating the appropriate SQL expressions for completing units of work.

Usage

A session object holds a `DatabaseAccessor` object and an instance of the `DescriptorSystem` subclass associated with your application. These must be set when the session object is created.

The API for class `DescriptorSystem` provides the simplest way to create a session object:

```
aSession := MyDescriptorSystem sessionForLogin: aLogin
```

Using `sessionForLogin:` and an initialized `Login` object, you can obtain a session object.

Alternately, you can build the session object explicitly, using `Login` and `DatabaseAccessor` objects, and the descriptor system class, e.g.:

```
aSession := GlorpSession new.
aSession system: (MyDescriptorSystem forPlatform: myLogin database).
aSession accessor: accessor.
```

When you set the descriptor system using `system:`, the session makes a run-time copy for use in the duration of the session. If you make any changes to the behavior of the descriptor system class, they will not be reflected in any active sessions. For that, you must create a new session object.

Note that if you are using connection pooling, you must build the session object explicitly rather than using `sessionForLogin:`.

Chapter

3

Active Record

Topics

Active Record is a pattern for object-relational mapping, first described by Martin Fowler. It makes it easy to persist a domain model, with an absolute minimum of metadata. The pattern uses a set of standard conventions to map domain classes to database tables, so that a single instance of a class represents a row in the corresponding table.

The Active Record framework provides all of the machinery for reading, writing, and updating objects as rows in database tables. In this respect, Active Record differs from the "classes as storage managers" pattern.

While Active Record provides a very simple framework for managing persistence, in practice most Glorp applications will either use Atlas and/or hand-code their model of the database.

Loading Active Record

To load support for Active Record into your VisualWorks image, open the Parcel Manager (select **System > Parcel Manager** in the Launcher window), select the **Directories** tab, the **glorp** directory, and load the `Glorp-ActiveRecord` parcel by double-clicking on the desired items in the upper-right-hand list of the Parcel Manager.

Domain Classes and Active Record

When using Active Record, the domain classes in your application should be defined as subclasses of `Glorp.ActiveRecords.ActiveRecord`.

While domain classes in Glorp may need an instance variable called `id`, to hold pimary key information, this is not required for domain classes that are persisted with Active Record. This variable has already been defined as part of the superclass `ActiveRecord`, and is thus inherited behavior.

Modeling for Active Record

Active Record provides a mechanism to automatically map between Smalltalk domain classes and database tables, by inferring the attributes of the columns. Tables are mapped to classes and, in the case of simple data types (e.g. `String`, `ByteArray`, `Float`, `Integer`, etc.), their columns are automatically mapped to instance variables in the classes.

When building an application that uses Active Record to store its domain model, the main issue facing the developer is whether the automatic mechanisms can perform all of the mapping, or whether it is necessary to customize the database schema, writing some of the mappings yourself. This depends primarily upon the complexity of the domain model. You can evaluate this by reviewing the mapping rules described below.

The Active Record pattern has slightly different requirements for the design of your domain model classes. These are explored in the topics that discuss the creation of your domain model. The Active Record implementation for Smalltalk is expressed as an extension to

Glorp, and as with any Glorp application, a descriptor system class needs to be defined to encapsulate the domain model metadata. In this way, Active Record requires that developers create fewer methods in the descriptor system class.

Mapping Rules

The mapping of class and variable names is handled automatically by the Active Record class `Inflector`. This is a utility class, and your application doesn't need to invoke it directly. During mapping, names are considered to be case insensitive, and plural forms are used by default. For example, a domain class named `Person` is mapped to a table named `PEOPLE`. If the name has several parts (separated by underscores or using Camel case), only the last is pluralized. If the mapping mechanism cannot find a table with a pluralized name, it tries again with a singular form. If that fails, an error is raised that the table cannot be found.

If you wish to change the default mapping behavior, class `Inflector` includes accessor methods to control the pluralization of table names (`pluralTableNames` and `pluralTableNames:`), and which form is attempted first. Either way, both singular and plural forms will be attempted during mapping.

If your domain model requires a complex schema, or if you are modelling an existing database schema that doesn't match the Active Record conventions, it is necessary to build the descriptor system class by hand, as the framework cannot generate all mappings dynamically. Active Record works for direct mappings, `One-To-One`, and `To-Many` mappings. In the latter cases, the framework requires a foreign key constraint already defined in the database.

A database schema is considered "complex" if it includes column names which do not match instance variable names, or tables used in `Many-To-Many` relationships, but for which there are no corresponding domain classes. Relationships between tables that do not include foreign key constraints are also considered complex, as are combined foreign key constraints (e.g.: using two columns to map to another table instead of just one). If your model subclasses an abstract class, including variables, your schema is likewise complex. In each of these cases, the schema class needs to be customized using Glorp mappings.

On the Smalltalk side, if your domain classes include instance variables holding collection objects (e.g., `Set`, `Dictionary`, or `SortedCollection`), you'll need to implement custom mappings in the schema. For collections, and for any `To-Many` relationship, Active Record assumes a mapping to an `OrderedCollection`. The mapping is created from the foreign key constraint, but it defaults to an `OrderedCollection`. If you wish to use a different collection class, you must change this by adding methods to the schema class. If, for example, you want your domain class to use a `Set`, then this needs to be indicated in the descriptor class.

Saving Objects

When using Active Record to persist objects in the database, several patterns are available.

Without Active Record, the usual method for saving objects with Glorp is to use `register:` inside a unit of work. With Active Record, you should send `bePersistent` directly to the domain object, followed by `commitUnitOfWork`.

For example, to create a new instance of class `Customer` and save it in the database, use the following pattern:

```
aCustomer := Customer new bePersistent
  firstName: 'Richard';
  lastName: 'Jones';
  yourself.
aCustomer commitUnitOfWork.
```

Sending `bePersistent` has the effect of registering an object, by marking it to be saved when the current unit of work is finished. If there is no active unit of work, one is started. While many implementations of the Active Record pattern use a method called `save`, Glorp has slightly different semantics (e.g., true object-level transactions), and thus opts for the more literal `bePersistent`.

Deleting Objects

When using Active Record to remove saved objects from the database, you may either send `delete` directly to the domain object or use `delete:` with a session object, followed by `commitUnitOfWork`, e.g.:

```
aCustomer delete.
aCustomer commitUnitOfWork.
```

Or:

```
aSession delete: aCustomer
aCustomer commitUnitOfWork.
```

The delete operation presupposes that the `id` of the domain object is correct. Whether you are using Glorp with Active Record or not, the semantics of `delete:` are the same.

Reading Objects

When using Active Record, you can read objects from the database by using either the Active Record or the standard Glorp API. For the former, use the methods `find:` and `findAll`. These messages may be sent to any Active Record domain object.

For example, to read all `Customer` objects:

```
customers := Customer findAll.
```

In effect, `findAll` is the same as using `read:` with the Glorp session object.

To read a specific object:

```
aCustomer := Customer find: 1.
```

The method `find:` expects a numeric primary key as its parameter, and returns the specified object. Generally, this key is held in the instance variable `id`.

Active Record also provides a pattern for reading objects according to specified criteria. For example:

```
Person findWhere:
 [:each |
 (each name = 'John Doe') AND: [each address houseNumber = 1000]].
```

This queries the database for all objects of class `Person` satisfying the specified block. Note that this must be a query block, meaning you can't execute arbitrary Smalltalk code.

Using Queries with Active Record

Like the Glorp API for queries, there are two ways to fetch objects from the database using Active Record. The simplest way is to use `findAll` or `find:` to read instances of a given class. The other approach is to assemble a `Query` object and then execute it. The advantages of the second approach are that you can create more elaborate queries and then reuse them easily.

For example, to query for all instances of class `Person`, and order them by their last name:

```
people := Person queryAll orderBy: #lastName; executeIn: aSession.
```

By sending successive `orderBy:` messages to the query object, you may assemble complex queries. For example:

```
aQuery := Address queryAll.
aQuery
 orderBy: [:each | each street];
 orderBy: [:each | each number].
addresses := aQuery executeIn: aSession.
```

To query for a single object, by its primary key:

```
aPerson := Person query: 1; executeIn: aSession.
```

To query using a query block:

```
aQuery := Person
  queryWhere:
```

```
[:each |
(each name = 'John Doe') AND: [each address houseNum = 1000]].
```

Note that the query block has special semantics, which differ in some important respects from those of a standard Smalltalk `BlockClosure`.

Updating Objects

When using Active Record to read and update objects in the database, use `commitUnitOfWork`.

For example, To read a specific object, and update it in the database:

```
aCustomer := Customer find: 1.
aCustomer phoneNumber: newNumber.
aCustomer commitUnitOfWork.
```

Here, a unit of work is created when the `Customer` object is modified, and the object is updated when the unit of work is completed. Note that `commitUnitOfWork` may cause other objects to be saved as well.

If for any reason you decide abort a unit of work, you can use `rollbackUnitOfWork`, e.g.:

```
aCustomer := Customer find: 1.
aCustomer phoneNumber: newNumber.
...
aCustomer hasValidPhoneNumber
 ifTrue: [aCustomer commitUnitOfWork]
 ifFalse: [aCustomer rollbackUnitOfWork].
```

Note that `rollbackUnitOfWork` rolls back the changes for *all* the objects in the current unit of work. You can also rollback the changes for a specific object by sending `rollback`, which has the effect of removing the receiver (a domain object) from the current unit of work.

Chapter

4

Atlas

Topics

The Atlas is a set of utilities that can be used to help generate Glorp descriptor systems from existing database schemas. The Atlas is conceptually similar to Active Record but capable of reading more general schemas.

The VisualWorks Altas may be used together with the ObjectStudio Mapping Tool. ObjectStudio's modelling and mapping tools provide a powerful UI with which to drive the whole Atlas. The VisualWorks Atlas is far more limited, but it is possible to develop Glorp models in ObjectStudio and then load them in VisualWorks.

Active Record can build a descriptor system for schemas that fit the strictly-defined Active Record pattern. Atlas, by contrast, can generate descriptor system code for most cases. If it encounters a schema so involved or ambiguous that it cannot generate a matching descriptor, it includes commented halts at the points which cannot be disambiguated. This approach — 95% automatic generation, 5% coding perspiration — can greatly speed Glorp-ifying existing legacy applications, especially if the schema is ill-documented.

Loading the Atlas

To load the Atlas into your VisualWorks image, open the Parcel Manager (select **System > Parcel Manager** in the Launcher window), select the **Directories** tab and the **glorp** directory, and load whichever of the GlorpAtlas parcels your task requires (by double-clicking on the desired items in the upper-right-hand list of the Parcel Manager.)

Load these parcels in VisualWorks only; if you are working in ObjectStudio, load the Mapping Tool.

Domain Classes and Atlas

When using the Atlas, the domain classes in your application need not be subclasses of Glorp.PersistentObject, any more than for hand-coded Glorp domain classes. It is, however, an easy superclass to start from, and the tools offer it by default.

Although nothing in Atlas requires that you use the Active Record pattern, equally nothing prevents you from choosing to generate domain classes with the superclass ActiveRecord and descriptor systems with the superclass ActiveRecordDescriptorSystem. If you choose one of these in the UI, the tool will prompt you to choose the other, for consistency.

Using the Atlas

If a Glorp model has been prepared using the ObjectStudio Modelling and Mapping tools, and saved to a parcel or package, loading that component into a VisualWorks image requires that you first load the parcel GlorpAtlasSystemsInVW. Nothing more is required if you are using a model prepared in ObjectStudio.

GlorpAtlasSystemGeneration

This supports generating a subclass of DescriptorSystem, with classFor<Name>:, descriptorFor<Name>: and tableFor<Name>: methods, from a descriptor system. Typically, this will be a descriptor system created when a database's metadata is read by GlorpActiveRecord or GlorpAtlasSystemGeneration.

This utility can be used programmatically or driven by `GlorpAtlasUI`. The programmatic API is:

```
WriteAtlasHelper system: aDescriptorSystem.
aDescriptorSystem
 writeAtlas: aClassNamePrefixString   "see #atlasNameExtension"
 in: anExistingNameSpace              "see #classBase"
 package: aNewOrExistingPackageName
```

This utility does not yet handle all cases in all possible schemas. Use and experience will add to what it can handle, and improve the warnings that are offered for cases it cannot. Post-generation hand-editing is essential in such cases, and useful in most cases: generation is meant to speed development of a descriptor system subclass for an existing database, not wholly to replace hand-coding.

GlorpAtlasClassGeneration

This extends the `GlorpActiveRecord` utilities. Like Active Record, it automatically reads the schema to create a descriptor system that maps between them, but it can read from schemas that do not adhere to the Active Record pattern, and it can generate classes that match the database tables. Invoke it as follows:

```
GlorpMetaProcess
 fromTablesIn: aLogin
 schema: aSchemaNameString
 generateClassesIn: anExistingNameSpaceName
 packageName: aNewOrExistingPackageName
 descriptorSystemClass: aNewOrExistingClassName.
```

This method returns the generated descriptor system. It checks the supplied schema name against the list returned by:

```
session := MetadataDescriptorSystem forLogin: aLogin.
session login.
session platform readSchemasForSession: session.
```

and throws an error if it cannot find a match. If it cannot create a mapping for a relationship, it writes a message to the Transcript.

GlorpAtlasUI

This is a simple VisualWorks tool that kickstarts fresh application development from a legacy database application. It lacks the full power of the ObjectStudio mapping tool (its UI employs lists and checkboxes, not a graphical model of a schema), and of course it has none of the mapping tool's integration with ObjectStudio's modeling tool. However, it does provide VisualWorks' users a UI with which they can get started on a Glorp schema for a legacy database if they lack the ObjectStudio suite (e.g. if they are on a non-Windows platform).

To open it, select **Tools > Glorp Atlas UI** in the Launcher window.

A simple tabular UI drives the Atlas to generate domain classes from the candidates the user selects, and a descriptor system from database tables.

Successive pages in the tool let you choose:

- The login to the database
- The tables of interest, and the proposed domain class names for them

A final post-generation page provides example Glorp code for connecting to the database.

Loading the AtlasUI also adds a page to the Settings tool, where default superclasses for generated domain classes and descriptor systems can be assigned.

Chapter

5

Modeling a Database

Topics

Glorp may be used to model an existing set of tables in a relational database or to create new ones. In both cases, your application needs a *descriptor system class* to represent the database model and the mappings between Smalltalk objects and relational tables. This class is not part of your domain model per se, but rather contains metadata for mapping your domain model to database tables.

Typically, a single descriptor system class is used for the application's domain model. There may be multiple, related classes in the domain model, all of which are modelled using a single descriptor system. I.e., all metadata is encapsulated in this class.

The descriptor system class provides behavior that specifies the names of all tables in the data model, the classes to which they correspond, the structure of each table, and the instance variables to which they are mapped. This behavior is expressed by adding instance-side methods that specify all the mappings.

Whether you wish to persist a Smalltalk domain model in a database, or to begin from an existing set of database tables, modeling a database involves the following basic tasks:

1. Create a descriptor system class
2. Add methods to model the database tables
3. Add methods to build descriptors
4. Add methods to build class models

If you are using Active Record to persist your domain model, in most cases only the first two steps are required.

Creating Domain Classes

The objects that Glorp maps to database tables are represented using domain classes. For example, the instances of a class named `MyCustomer` might be mapped to rows in a table named `my_customer`.

As a rule, database columns correspond to instance variables in a domain class. If your table includes a column used as a primary key, you should likewise include that as an instance variable in the domain class. In this case, it is common to have a column called `id`.

With respect to accessor methods in your domain classes, Glorp requires that any value obtained from a getter can be used as an argument to the corresponding setter method, without affecting the result of a subsequent invocation of both methods. Further, these methods should not perform any side effects or other actions, i.e., they simply get and set the value of the instance variables.

Naming Conventions

Glorp respects some common naming conventions of RDBMS, but also tries to allow developers flexibility.

For example, table and column names are frequently set in lower case, with words being separated by underscores. In Smalltalk, of course, most names use camel case and underscores are discouraged.

Glorp allows both conventions, but has two requirements: first, that table, column, and constaint names must begin with a letter. Following this letter, names can use any combination of alphanumeric characters and underscores. They cannot begin with an underscore, though.

Second, Glorp requires that constraint names must use the same name as the table to which they are applied, appended with a particular suffix. E.g., for a table called `Customer` a primary key applied to one column would be called `Customer_PK`.

Domain Classes and Active Record

When using Active Record, the domain classes in your application should be defined as subclasses of `Glorp.ActiveRecords.ActiveRecord`.

While domain classes used with Glorp may need an extra instance variable called `id`, to hold pimary key information, this is not required for domain classes that are persisted using Active Record. This variable has already been defined as part of the superclass `ActiveRecord`, and is thus inherited behavior in your domain classes.

Creating a Descriptor System

When using Glorp, the domain model of your application must be modelled using a *descriptor system*. This is a class that you create as part of your application, to gather together all the descriptors for your domain model (i.e., its schema), using a number of instance-side methods. The descriptor system must be defined as a subclass of `DescriptorSystem` (or, if you are using Active Record, a subclass of `ActiveRecordDescriptorSystem`).

The behavior of the descriptor system models the classes in the domain model, and specifies the database tables to which they should be mapped. Each class in the domain model and each corresponding table in the database are represented by a distinct method in the descriptor system class. This single class, then, contains all the metadata and mapping specifications needed to use your application with Glorp.

To create a descriptor system class, use the System Browser to define a new class as follows:

```
MyAppNameSpace defineClass: #MyDescriptorSystem
 superclass: #{Glorp.DescriptorSystem}
 indexedType: #none
 private: false
 instanceVariableNames: ''
 classInstanceVariableNames: ''
 imports: 'Glorp.*'
 category: 'MyApplication'
```

Note that the descriptor system class should be defined in the same name space as your domain model (e.g., `MyAppNameSpace.*`), and it should import the `Glorp.*` name space.

At runtime, Glorp creates an instance of this descriptor system class for each database session, and uses it to manage the mechanics of object-relational mapping. Glorp handles all the details, and thus you only need to specify the descriptor system correctly once. Of course, if you change your domain model, the descriptor system may need to be changed as well.

To express the mapping between database tables and Smalltalk objects we need to add a number of methods to this class, implementing part of the API for `DescriptorSystem`. These methods provide behavior for:

1. Modelling the database tables
2. Building descriptors for all the classes in the domain model
3. Building class models

If you are using Active Record to persist your domain model, only the first of these steps is required. The Active Record framework dynamically generates the descriptors and class models. If necessary, you can add methods to the descriptor class, which override the dynamic mappings.

If you are using Active Record to work with pre-existing tables in the database, you don't need to implement any of these methods, as long as your domain classes and database tables follow the Active Record naming conventions.

If you are not using Active Record, or if you have a very complex model and need to build the metadata by hand, the methods of descriptor system class are organized as follows. All of these methods have standardized names (see below), each expecting a parameter object that is filled out with metadata. There are three basic method patterns:

classModelForNAME:

Expects an instance of `ClassModel` as a parameter, which gets filled out with information about the instance variables of class NAME. For each class in the domain model, there is a corresponding classModelForNAME: method.

tableForNAME:

Returns information about the columns in table NAME.

descriptorForNAME:

Defines the mapping between the instance variable in class NAME and one or more tables as defined in various tableForNAME: methods. Expects an instance of Descriptor as a parameter.

The VisualWorks Tools include "framework-oriented refactorings" such that renaming a class offers to also rename any methods whose selectors match descriptorFor<ClassName>: or classModelFor<ClassName>.

The descriptor class also includes two other methods, which idenitify all of the tables and classes in the domain model:

allTableNames

Return an Array of strings, of all the tables that belong to the model.

constructAllClasses

Return an OrderedCollection of strings, naming all of the classes that belong to the domain model.

Step-by-step instructions for writing these methods are provided in additional topics.

Modeling Database Tables

A descriptor system class must implement several methods that serve to model a database table. Typically, a single descriptor system is used for the application's domain model. There may be multiple, related tables in the data model, all of which are modelled using a single class.

As a simple example, consider a table called Customer that has three columns: a cust_id column containing an integer, followed by first_name, and last_name, two columns containing Strings up to 64 characters. Assume that cust_id is the key for each Customer object. This table could be created using the following SQL:

```
CREATE TABLE Customer(
 cust_id integer CONSTRAINT customer_PK PRIMARY KEY,
 first_name varchar(64),
 last_name varchar(64));
```

To provide Glorp with a specification of this table, two methods are required in the descriptor system class:

1. In the accessing protocol, add the following method:

   ```
   allTableNames
    ^#('Customer')
   ```

   This method must return an Array of Strings, one for the name of each table to be mapped to a Smalltalk domain class.

   Depending upon the database vendor and configuration, table names may be case insensitive, but Smalltalk strings and method names are case sensitive. You should ensure that all references to tables in the descriptor system class use the same case as the table name.
2. For each table in the allTableNames array, we need a method in the tables protocol called tableForNAME: where NAME is replaced with the table name. E.g., the method for our example looks like this:

   ```
   tableForCustomer: aTable
    (aTable
     createFieldNamed: 'cust_id'
     type: accessor platform int4) bePrimaryKey.
    aTable
     createFieldNamed: 'first_name'
     type: (accessor platform varChar: 64).
    aTable
     createFieldNamed: 'last_name'
     type: (accessor platform varChar: 64).
   ```

Note that we do not use SQL to define the table, but rather Smalltalk expressions. This enables Glorp to generate the appropriate SQL for any database we might choose. In general, the name and type each column must be declared.

Note that if your domain classes change (e.g., name, names or contents of instance variables), or if the definition of any database tables used by your application change, then the descriptor system must likewise be revised appropriately.

Modeling Tables Containing Primary Keys

When working with database tables defined by another application, you may need to explicitly declare that one column is a primary key. For example:

```
tableForPEOPLE: aTable
 aTable
  createFieldNamed: 'first_name'
  type: (platform varChar: 50).
 (aTable
  createFieldNamed: 'last_name'
  type: (platform varChar: 50)) bePrimaryKey.
```

Alternately, you can use the message `addAsPrimaryKeyField:`, e.g.:

```
tableForPEOPLE: aTable
 | keyField |
 aTable
  createFieldNamed: 'first_name'
  type: (platform varChar: 50).
 keyField := aTable
      createFieldNamed: 'last_name'
     type: (platform varChar: 50).
 aTable addAsPrimaryKeyField: keyField.
```

For multiple-column primary keys, use the `bePrimaryKey` message for each column spanned by the key, e.g.:

```
tableForPEOPLE: aTable
 (aTable
  createFieldNamed: 'first_name'
  type: (platform varChar: 50)) bePrimaryKey.
 (aTable
  createFieldNamed: 'last_name'
  type: (platform varChar: 50)) bePrimaryKey.
```

Modeling Tables Containing Foreign Keys

To model a database table that includes a foreign key,* you must declare the key in a `tableForNAME:` method. This is necessary whenever a class in your domain model includes an instance variable that holds another complex (i.e., non-primitive) class.

For example, if we have a `Customer` class in our domain model which holds an instance of class `Address` in a variable named `address`, then we need to add a field to the `Customer` table that references the `Address` table using a foreign key, e.g.:

```
addressId := aTable
    createFieldNamed: 'address_id'
    type: platform int4.
aTable
 addForeignKeyFrom: addressId
 to: ((self tableNamed: 'Address') fieldNamed: 'id').
```

Regardless of whether your application takes responsibility for creating the database tables or not, the descriptor system class must include the `allTableNames` and `tableForNAME:` methods. Finally, to complete the descriptor system, you must also add methods that define a `ClassModel`.

* See Foreign Key Mapping pattern p. 236-247 in Fowler [2003].

Using Sequences

Tables in relational databases use primary keys as unique identifers for each row. In Smalltalk, by contrast, instances of your domain classes are unique by virtue of the semantics of objects. To maintain this uniqueness when persisting your domain objects in a database, you typically use a primary key, stored in an instance variable.

The database server can create a unique value for this primary key each time you add a new row to the database. This is known as a *sequence*. To use this feature, your application needs a little extra code in the descriptor class method that specifies the table.

For example, let's say we want to automatically generate a primary key for each instance of class `Person`. The table definition method would look as follows:

```
tableForPEOPLE: aTable
 (aTable
  createFieldNamed: 'id'
  type: platform sequence) bePrimaryKey.
 (aTable
  createFieldNamed: 'first_name'
  type: (platform varChar: 50)).
 (aTable
  createFieldNamed: 'last_name'
  type: (platform varChar: 50)).
```

The variable named `id` is used to hold the primary key. The descriptor for class `Person` doesn't require anything special, aside from a mapping for the `id` field:

```
descriptorForPerson: aDescriptor
 | personTable |
 personTable := self tableNamed: 'PEOPLE'.
 aDescriptor table: personTable.
 (aDescriptor newMapping: DirectMapping)
  from: #firstName
  to: (personTable fieldNamed: 'first_name').
 (aDescriptor newMapping: DirectMapping)
  from: #lastName
  to: (personTable fieldNamed: 'last_name').
 (aDescriptor newMapping: DirectMapping)
  from: #id
  to: (personTable fieldNamed: 'id').
```

When persisting domain objects in a table that uses a sequence, the primary key is set automatically. You can save objects with the usual code pattern:

```
session inTransactionDo:
 [person := Person first: 'Groucho' last: 'Marx'.
 session register: person].
```

Here, if the id variable in the Person object is nil when it is registered with the session, Glorp will fetch the next value in the sequence and set the column id to that. Then object is then added to the table.

If you create a new instance of Person, and set the value of id to something other than nil, the Glorp framework will again add it, though this rather defeats the purpose of using a sequence. If you read an object from the database, it will have an assigned id. If you then modify the object and commit the unit of work, it will be save the object in the correct row of the table.

Managing Tables and Sequences

If your application is responsible for creating the database tables, Glorp can automatically handle the creation of sequences as well. However, depending upon the way you manage the tables, it may be necessary to explicitly create the sequences on certain database platforms (e.g. Postgres, Oracle).

If you use the methods createTables, dropTables, or recreateTables, Glorp manages sequences automatically. However, if you use createTable:ifError:, your application needs to explicitly create all tables and associates sequences, as follows:

```
[session system platform areSequencesExplicitlyCreated
 ifTrue:
  [session system allSequences do:
   [:each |
   accessor createSequence: each
    ifError: [:error | Transcript show: error messageText]]].
session system allTables do:
 [:each |
 accessor createTable: each
  ifError: [:error | Transcript show: error messageText]]].
```

Here, the method areSequencesExplicitlyCreated returns true for those platforms that require explicit handling of sequences.

Likewise, if you use dropTable:ifError:, you need analogous code that sends dropSequence:.

Using Relative Fields

Glorp manages the risk of concurrent updates to the same data by using transactions, locks or optimistic locking (see Locking Objects in the Database). Optimistic locking requires that a row be reloaded if its contents have changed since Glorp last read it. If a high-speed multi-user application needs to update just particular columns, such as inventory or dollar totals, in a domain where optimism is unjustified — i.e. concurrent clashing updates are likely — a faster approach is to make the field update the relative change in value. To suuport this type of high-speed application, Glorp provides a feature called *relative fields*.

If a field's type is `GlorpRelativeType` (wrapping an underlying type which is usually a subclass of `GlorpNumericType`), then Glorp computes the field's change in value since it was read, and updates the target column value by that amount. Thus, if several users are looking at the same row, the combined total value in the database after two changes is correct, regardless of their order or whether either user refreshed the object before updating it.

For example, if a relative field was read with value `100` and a user changed it to `110`, then committed their unit of work, the generated SQL will look like:

```
UPDATE inventory SET qty = qty + 10 WHERE ... RETURNING qty
```

The `RETURNING` clause sets the final value of the field on the instance in the image. This happens in the same round trip, thereby providing the desired performance while allowing applications to be in synch with the database after an update.

Note: Not all database platforms support `RETURNING`: i.e., a platform that returns `false` to `supportsReturningUpdatedValues` will not set the new value, so use relative types with caution on such platforms, either discarding written objects, or explicitly refreshing them, if they need to assume the absolute correctness of values after update.

In cases where a table has both lock key and relative fields, the optimistic locking behavior will occur as usual unless it happens that only the relative field(s) values have changed in an update. In that case, the relative field(s) are updated and the lock key(s) are unaffected. We believe this to be the correct pattern in this probably rare case, but any users who combine the two approaches in one table are advised to reflect on what it means for their application.

Examples of the use of relative fields can be found in the `GlorpTest` parcel: see `GlorpRelativeFieldTest` and browse senders of `beRelative`.

Defining Column Types

A Glorp descriptor system class implements several methods that serve to model a database table, including declarations for the names and types of the columns. Each table is represented as an instance of `DatabaseTable`, and each column as an instance of `DatabaseField`.

To build a `DatabaseTable` object, the descriptor system class includes a `tableForNAME:` method for each table, e.g.:

```
tableForCustomer: aTable
 (aTable
  createFieldNamed: 'cust_id'
  type: accessor platform int4) bePrimaryKey.
 aTable
  createFieldNamed: 'cust_name'
  type: (accessor platform varChar: 255).
```

The method `createFieldNamed:type:` adds a `DatabaseField` to the table, one for each column. To specify the type of the column, the appropriate `DatabasePlatform` class provides a group of methods in the `types` protocol. These correspond to familiar SQL types such as `VARCHAR`, `INT`, `DATE`, etc.

The method `bePrimaryKey` specifies that the column should be a primary key.

Using Column Type Modifiers

When specifying the attributes of a database table in a Glorp descriptor system class using the tableForNAME: method, sometimes you must use type modifiers. For example, in this method:

```
tableForCustomer: aTable
 (aTable
  createFieldNamed: 'cust_id'
  type: accessor platform int4) bePrimaryKey.
  ...
```

The message bePrimaryKey acts as a type modifier for the DatabaseField object that represents the cust_id column.

The following modifiers are part of the API to class DatabaseField:

beNullable: aBoolean

Send beNullable: false to require that the column can never be set to NULL. By default, this is true, i.e., the column can be NULL.

isUnique: aBoolean

Send isUnique: true to require that all values in the column must be unique. By default, this is false.

defaultValue: anObject

If the instance variable mapped to the column is nil, set the default value of the column to anObject. Note that this does not use the SQL mechanism for defaulting; instead, Glorp inserts the default before transferring the object to the database.

beLockedKey

Specify that the column is a locking key.

bePrimaryKey

Specify that the column is a primary key.

beRelative

Specify that the column is a Relative Field. For details, see the discussion in Using Relative Fields. This can be sent to the field or to its type. It wraps the underlying type — which must be an appropriate Glorp type for the column in the database — with a wrapper that controls how update SQL is generated.

It is possible to cascade type modifiers, as follows:

```
(aTable createFieldNamed: 'dept_code' type: (platform int))
 defaultValue: 5;
 beNullable: false.
```

Using Pseudo Variables

Pseudo variables are variables that do not exist in a domain class, but can be used in a query about the class. The general use-case for these is that you want to reference a database column in a query, but you don't want it mapped into a Smalltalk object. This can increase performance, as you may perform complex queries without the overhead of always reading/writing from the database.

For example, let's say you want to fetch a collection of `Customer` objects based upon the number of orders and returns that each has made. The `Customer` table contains columns for these values (e.g. `num_returns` and `amount_ordered`), but your application only needs to query against them. In this situation, you can use pseudo variables.

To do this, first you need to define the pseudo variables in a descriptor:

```
(aDescriptor newMapping: DirectMapping)
 fromPseudoVariable: #numberOfReturns
 to: (personTable fieldNamed: 'num_returns').
(aDescriptor newMapping: DirectMapping)
 fromPseudoVariable: #amountOrdered
 to: (personTable fieldNamed: 'amount_ordered').
```

Here, the method `fromPseudoVariable:to:` defines the pseudo variable (you can also use the convenience API: `beForPseudoVariable`) and maps it to the named database column.

At this point, you can compose a complex query using the pseudo variables:

```
session
```

```
read: Customer
where:   [:each |
   each numberOfReturns > 10 & (each amountOrdered < 1000)].
```

Pseudo variables can also be used for aliasing. That is, the database table might call primary keys `primary_key`, while you wish to use the more abbreviated `id`. Alternately, they can be used to create different quries against columns that are already mapped in a different way.

For example, if the `Customer` table includes an address that includes a city and state, but you want to write queries against the state, then you can define a pseudo variable for `state` which uses a double join, e.g.:

```
session read: Customer orderBy: #state.
```

This example illustrates a more general point: pseudo variables can be used to maintain the ordering of a collection. By nature, relational databases don't have a concept of this. That is, when querying the rows in a table, they may be returned in an arbirary order. By contrast, many collection classes in object-oriented languages have an implicit order, and it's not unusual for algorithms to rely on that order.

In practice, you can define a column that holds a value for ordering the rows (e.g. `order`), use the `orderBy:`message when defining a mapping for that field, and that will automatically write the field and later read things back in the correct order. On the Smalltalk side, however, we don't really want to hold this value in the domain class, worry about updating it, etc., so there is no mapping to an instance variable. Using a pseudo variable, though, we could still query against that column, or put an `orderBy:` on it.

Building a ClassModel

A descriptor system class must implement several methods that assist when mapping the classes of your application's domain model to database tables, detailing how their instance variables relate to database columns.

Some of this information is captured in a `ClassModel` object. Glorp manages `ClassModels` internally, but the developer needs to define some methods that express their composition.

Note: these methods are not required when using Active Record.

For example, to create a `ClassModel` that may be used to map a Smalltalk class called `Customer`, add the following two methods to your descriptor system class:

1. First, define a method named `constructAllClasses` (in the `initialization` protocol) that returns a `Collection` of all the domain classes to be mapped by this descriptor system. E.g.:

   ```
   constructAllClasses
    ^(super constructAllClasses)
     add: Customer;
     yourself
   ```

   Although the method name suggests that the classes will be created, this method only needs to return a `Collection` of class objects.
2. Next, we add a method that creates a `ClassModel` object. Again, one method is required for each domain class. The method must be called `classModelForNAME:`, using the name of the class.

   The `ClassModel` object holds metadata that describes a class, primarily its instance variables. However, in a simple example like the one we are discussing here, this method doesn't itself need to provide a `ClassModel`, as Glorp handles this automatically. A more detailed illustration of how to create a `ClassModel` is given elsewhere.

   For the purposes of our example, then, the method can look like this:

   ```
   classModelForCustomer: aClassModel
    aClassModel newAttributeNamed: #id.
    aClassModel newAttributeNamed: #firstName.
    aClassModel newAttributeNamed: #lastName.
   ```

If you are modelling a class that includes instance variables that hold complex objects (e.g., a `BankAccount` class that holds onto an

Address object), you need to specify this explicitly when building its ClassModel.

For example:

```
aClassModel newAttributeNamed: #address type: Address.
```

The method newAttributeNamed:type: should be used whenever you use a OneToOneMapping.

Once the DescriptorSystem class is complete, you have all the pieces needed to connect to the database, perform queries and insert objects.

Building a Descriptor

A descriptor system class must include methods that describe how each domain class gets mapped to a database table. This is done using Descriptor objects, which hold mapping objects for each instance variable of the domain class.

Application developers do not need to manage descriptor objects directly. Rather, Glorp manages the descriptor objects internally, and the developer need only add some methods to express the mappings held by the descriptor for a given Smalltalk domain class. At runtime, Glorp invokes these methods to build descriptors as necessary.

To assemble a descriptor for a domain class, add a method to the descriptor system class called descriptorForNAME:,where NAME is replaced with the name of the class. Each domain class that is modelled in the descriptor system class needs a descriptorForNAME: method.

As a simple example, consider a domain class named Customerthat we want to map to table with three columns: a cust_idcolumn containing an integer, followed by first_name, and last_name, two columns containing Strings up to 64 characters. Assume that cust_id is the key for each Customer object. This table could be created using the following SQL:

```
CREATE TABLE Customer(
cust_id integer CONSTRAINT customer_PK PRIMARY KEY,
first_name varchar(64),
last_name varchar(64));
```

The method to build a descriptor object for `Customer` would look like this:

```
descriptorForCustomer: aDescriptor
 | table |
 table := self tableNamed: 'Customer'.
 aDescriptor table: table.
 (aDescriptor newMapping: DirectMapping)
  from: #id
  to: (table fieldNamed: 'cust_id').
 (aDescriptor newMapping: DirectMapping)
  from: #firstName
  to: (table fieldNamed: 'first_name').
 (aDescriptor newMapping: DirectMapping)
  from: #lastName
  to: (table fieldNamed: 'last_name')
```

For the purposes of this example, this method creates a `DirectMapping` between each instance variable in the class and a corresponding database column.

If you are modelling a class that includes instance variables that hold complex objects (e.g., a `Customer` class that holds onto an `Address` object), you need to specify this by using a `OneToOneMapping`.

For example:

```
(aDescriptor newMapping: OneToOneMapping)
 attributeName: #address.
```

In this situation, we must also indicate that the `ClassModel` object includes a primary key.

Using Mappings

A *mapping* defines a specific way to translate between a Smalltalk object and a database table column. Glorp provides a variety of different types of mappings, each being represented as a concrete subclass of class `Mapping`. The mapping class that you choose when creating a descriptor depends in part upon the class of the object that you wish to persist in the database.

There are three basic varieties of mapping:

DirectMapping

The simplest type of mapping, it maps directly between an instance variable that holds some primitive type (e.g., instances of `Number`, `String`, and `Boolean`), and a single column in a database table.

OneToOneMapping

This mapping represents a one-to-one relationship, e.g. from a `Person` object to an `Address` object. That is, for each `Person` there is only one `Address`. This mapping may be used for a variety of complex objects, such as collections and dictionaries.

Note that class `OneToOneMapping` has no methods, and exists mainly for backward compatibility, since a one-to-one relationship is the default for class `RelationshipMapping`.

ToManyMapping

This is a superclass for all mappings which represent a relationship between one object and a collection of others. It isn't an abstract superclass, you can use this class directly, and you will get the same behaviour as if you were using `OneToManyMapping`.

Mappings are used when building descriptor objects in the `descriptorForNAME:` methods that belong to a descriptor system class.

Instances of class `Descriptor` are used to hold the mapping information for each persistent class in the domain model. Note that a descriptor is distinct from a descriptor system, the latter being a single class that holds all of the metadata for a given application. I.e., the descriptor system describes the domain model as a whole, while each descriptor object holds only the information for a single domain class.

Using a DirectMapping

When mapping between primitive types such as class Number, String, Boolean, and Timestamp, you should use a DirectMapping. This mapping indicates a direct correspondence between an instance variable in the domain model and a database column.

For example, to create a mapping between an account number in a BankAccount object and a database table, and then to add it to a descriptor:

```
(aDescriptor newMapping: DirectMapping)
 from: #accountNumber
 to: (table fieldNamed: 'ACCT_NO').
```

Here, the symbol #accountNumber is the name of an instance variable in class BankAccount, and the table object is an instance of DatabaseTable that is used when building the descriptor.

Using a OneToOneMapping

When mapping an instance variable that holds a more complicated object (e.g., one that has its own instance variables), generally you want to use a OneToOneMapping.

For example, if your domain model includes a BankAccount class, with an instance variable named address that holds an instance of class Address, it is typically represented in the database using two tables, with a foreign key in the BANK_ACCOUNTS table pointing to a primary key in the ADDRESSES table.

To express this model, we need the following basic methods in the descriptor system class:

```
allTableNames
 ^#('BANK_ACCOUNTS' 'ADDRESSES')

 constructAllClasses
 ^(super constructAllClasses)
  add: BankAccount;
  add: Address;
  yourself
```

The descriptor system class also includes methods for each table. To model the `ADDRESSES` table, it might look something like this:

```
tableForADDRESSES: aTable
 aTable createFieldNamed: 'street' type: (platform varChar: 50).
 (aTable createFieldNamed: 'city' type: (platform varChar: 50)).
 (aTable createFieldNamed: 'state' type: (platform varChar: 50)).
 (aTable createFieldNamed: 'zip_code' type: (platform varChar: 20)).
 (aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey
```

For setting up a `OneToOneMapping`, the important element is the use of `bePrimaryKey`.

To model the `BANK_ACCOUNTS` table, the code could be:

```
tableForBANK_ACCOUNTS: aTable
 | addressId |
 (aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey.
 (aTable createFieldNamed: 'number' type: (platform varChar: 24)).
 (aTable createFieldNamed: 'first_name' type: (platform varChar: 50)).
 (aTable createFieldNamed: 'last_name' type: (platform varChar: 50)).
 addressId := aTable
       createFieldNamed: 'address_id'
       type: platform int4.
 aTable
  addForeignKeyFrom: addressId
  to: ((self tableNamed: 'ADDRESSES') fieldNamed: 'id').
```

Here, the foreign key relation is established by the last two lines of the method.

Each class in the domain model also requries a descriptor, so we need to define the methods `descriptorForBankAccount:` and `descriptorForAddress:`. The latter is straight-forward:

```
descriptorForAddress: aDescriptor
 | table |
 table := self tableNamed: 'ADDRESSES'.
 aDescriptor table: table.
 (aDescriptor newMapping: DirectMapping) from: #street
  to: (table fieldNamed: 'street').
 (aDescriptor newMapping: DirectMapping) from: #city
  to: (table fieldNamed: 'city').
 (aDescriptor newMapping: DirectMapping) from: #state
 to: (table fieldNamed: 'state').
(aDescriptor newMapping: DirectMapping) from: #zipCode
 to: (table fieldNamed: 'zip_code').
(aDescriptor newMapping: DirectMapping)
 from: #id
 to: (table fieldNamed: 'id').
```

The descriptor for `BankAccount` is slightly more complex:

```
descriptorForBankAccount: aDescriptor
 | bankAccountTable |
 bankAccountTable := self tableNamed: 'BANK_ACCOUNTS'.
 aDescriptor table: bankAccountTable.
 (aDescriptor newMapping: DirectMapping)
  from: #accountNumber
  to: (bankAccountTable fieldNamed: 'number').
 (aDescriptor newMapping: DirectMapping)
  from: #firstName
  to: (bankAccountTable fieldNamed: 'first_name').
 (aDescriptor newMapping: DirectMapping)
  from: #lastName
  to: (bankAccountTable fieldNamed: 'last_name').
 (aDescriptor newMapping: DirectMapping)
  from: #id
  to: (bankAccountTable fieldNamed: 'id').
 (aDescriptor newMapping: OneToOneMapping)
  attributeName: #address.
```

At the end of this method, we use a `OneToOneMapping` to refer to the `Address` object. However, there is no explicit mention of class `Address`. That is specified in a different method.

Finally, the descriptor system class also includes methods that define the class models identifying each mapped instance variable:

```
classModelForAddress: aClassModel
 aClassModel newAttributeNamed: #id.
 aClassModel newAttributeNamed: #street.
 aClassModel newAttributeNamed: #city.
 aClassModel newAttributeNamed: #state.
 aClassModel newAttributeNamed: #zipCode.

 classModelForBankAccount: aClassModel
 aClassModel newAttributeNamed: #id.
aClassModel newAttributeNamed: #accountNumber.
aClassModel newAttributeNamed: #firstName.
aClassModel newAttributeNamed: #lastName.
aClassModel newAttributeNamed: #address type: Address.
```

Here, the last line of code indicates that the instance variable `address` in a `BankAccount` object is mapped to an instance of class `Address`. In this fashion, the methods in the descriptor system are used to establish one-to-one mappings. At runtime, the `address` variable will contain `Address` objects, as expected. Behind the scenes, proxies are used so that the address object is only fetched from the database when it is actually referenced.

A variant of the `OneToOneMapping` is `EmbeddedValueOneToOneMapping`. Use this for a one-to-one relationship, in which the target object is not stored in a separate table, but rather as part of the row of the containing object.

Using a ToManyMapping

When mapping an instance variable that holds a collection of objects (for example, an `OrderedCollection`), you must use a `ToManyMapping` or one of its subclasses. The two most common are `OneToManyMapping` and `ManyToManyMapping`. The choice between these variants of a `ToManyMapping` depends in part upon the types of queries you expect to run against the domain objects.

One-to-Many Mapping

To represent an object that contains a collection of other objects, a relational database would use a separate table for the collection, with a foreign key in each row that references its owning object (i.e., the reference points in the opposite direction from the convention in Smalltalk).

As an example, let's say we have class `Person` that includes an instance variable `emailAddresses` containing a collection of `EmailAddress` instances. To model this in the database, we would have two tables: `PEOPLE` and `EMAIL_ADDRESSES`.

The descriptor system class would need the following methods to model the database tables:

```
allTableNames
 ^#('PEOPLE' 'EMAIL_ADDRESSES')

 constructAllClasses
 ^(super constructAllClasses)
  add: Person;
  add: EmailAddress;
  yourself

 tableForPEOPLE: aTable
 (aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey.
 (aTable createFieldNamed: 'first_name' type: (platform varChar: 50)).
 (aTable createFieldNamed: 'last_name' type: (platform varChar: 50)).

 tableForEMAIL_ADDRESSES: aTable
 | personId |
 aTable createFieldNamed: 'user_name' type: (platform varChar: 50).
 (aTable createFieldNamed: 'host' type: (platform varChar: 50)).
 (aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey.
 personId := aTable createFieldNamed: 'person_id' type: platform int4.
 aTable addForeignKeyFrom: personId to: ((self tableNamed: 'PEOPLE') fieldNamed:
 'id').
```

Note that the `EMAIL_ADDRESSES` table includes a foreign key that references a row in the `PEOPLE` table.

Methods to create the descriptor objects are also required:

```
descriptorForPerson: aDescriptor
 | personTable |
 personTable := self tableNamed: 'PEOPLE'.
 aDescriptor table: personTable.
 (aDescriptor newMapping: DirectMapping)
  from: #firstName
  to: (personTable fieldNamed: 'first_name').
 (aDescriptor newMapping: DirectMapping)
  from: #lastName
  to: (personTable fieldNamed: 'last_name').
 (aDescriptor newMapping: DirectMapping)
  from: #id
  to: (personTable fieldNamed: 'id').
```

```
(aDescriptor newMapping: OneToManyMapping)
 attributeName: #emailAddresses.

descriptorForEmailAddress: aDescriptor
| table |
table := self tableNamed: 'EMAIL_ADDRESSES'.
aDescriptor table: table.
(aDescriptor newMapping: DirectMapping)
 from: #userName
 to: (table fieldNamed: 'user_name').
(aDescriptor newMapping: DirectMapping)
 from: #host
 to: (table fieldNamed: 'host').
(aDescriptor newMapping: DirectMapping)
 from: #id
 to: (table fieldNamed: 'id').
```

Note that most of the mappings use DirectMapping, but that descriptorForPerson: method also includes aOneToManyMapping with the name of the instance variable emailAddresses specified as a symbol.

The descriptor system class also needs methods to set up the class model objects, as follows:

```
classModelForPerson: aClassModel
 aClassModel newAttributeNamed: #id.
 aClassModel newAttributeNamed: #firstName.
 aClassModel newAttributeNamed: #lastName.
 aClassModel newAttributeNamed: #emailAddresses collectionOf: EmailAddress.

 classModelForEmailAddress: aClassModel
 aClassModel newAttributeNamed: #id.
 aClassModel newAttributeNamed: #userName.
 aClassModel newAttributeNamed: #host.
```

Here, the OneToManyMapping is specified using the message newAttributeNamed:collectionOf: in classModelForPerson:.

When a Person object is read from the database, it contains a collection of EmailAddress objects. E.g.:

```
aPerson := aSession
  readOneOf: Person
  where: [:each | each firstName = 'Sam'].
```

```
gmailAddresses := aPerson emailAddresses select: [:addr | addr host = 'gmail.com'].
```

Ordered Reading

It is also possible to specify the order in which objects are read from the database. This can either be done using a `Query` object, or by using the `orderBy:` message when defining the domain mapping. This tells the database server to sort the results, which may be faster than sorting them inside your application.

For example, to sort by in the example shown above, modify one method in the descriptor system as follows:

```
descriptorForPerson: aDescriptor
 | personTable |
 personTable := self tableNamed: 'PEOPLE'.
 aDescriptor table: personTable.
 (aDescriptor newMapping: DirectMapping)
  from: #firstName
  to: (personTable fieldNamed: 'first_name').
 (aDescriptor newMapping: DirectMapping)
  from: #lastName
  to: (personTable fieldNamed: 'last_name').
 (aDescriptor newMapping: DirectMapping)
  from: #id
  to: (personTable fieldNamed: 'id').
 (aDescriptor newMapping: OneToManyMapping)
  attributeName: #emailAddresses;
  orderBy: #userName.
```

The effect of this additional message is to sort the email addresses by their username, which is one column in the table of email addresses. As with queries, complex sorting rules can be created by sending multiple `orderBy:` messages.

Many-to-Many Mapping

The second type of `ToManyMapping` is the `ManyToManyMapping`. Like a `OneToManyMapping`, this is also used to model an object that contains a collection of other objects. The `ManyToManyMapping` should be used when your application needs to manipulate collections of both the containing and contained objects.

To illustrate with an example, let's say we have a bookstore application for tracking customers' orders. The domain model includes classes for customers and books: Customer and Book. Each customer can order any number of books, and any book may be on order by a number of different customers. The bookstore application needs to be able to display all customers that have ordered a particular book, and it needs to be able to display all books on order by a particular customer. For this, a ManyToManyMapping is required.

In the database, a many-to-many relationship is represented using an intermediate link table (sometimes called an association or tie table). For this example, it is called BOOKS_ON_ORDER. This table holds associations between rows in the CUSTOMERS and BOOKS tables. Each row in the link table represents a book order by a single customer. More specifically, it contains one column that refers to a row in the CUSTOMERS table, and another column that refers to a row in the BOOKS table. In total, then, this data structure requires three tables.

Defining the Domain Classes

For the domain model, the class definition for Customer is:

```
Smalltalk defineClass: #Customer
 superclass: #{Core.Object}
 indexedType: #none
 private: false
 instanceVariableNames: 'id firstName lastName booksOnOrder '
 classInstanceVariableNames: ''
 imports: ''
 category: 'My Application'
```

The instance variable booksOnOrder holds an OrderedCollection of Book objects. The variables firstName and lastName are intended to hold strings.

As a minimal definition, the Book class looks like this:

```
Smalltalk defineClass: #Book
 superclass: #{Core.Object}
 indexedType: #none
 private: false
```

```
instanceVariableNames: 'id title '
classInstanceVariableNames: ''
imports: ''
category: 'My Application'
```

The domain classes need a number of initialization and access methods, including basic accessors for all of the instance variables (which we'll omit for the sake of brevity).

Class `Customer` needs a class-side method to create an initialized instance:

```
first: firstNameString last: lastNameString
 ^self new setFirst: firstNameString last: lastNameString
```

The following instance-side methods are also required:

```
setFirst: firstNameString last: lastNameString
 firstName := firstNameString.
 lastName := lastNameString.
 booksOnOrder := OrderedCollection new.

 addBook: aBook
 booksOnOrder add: aBook

 books
 ^booksOnOrder
```

Class `Book` also needs a class-side method to create an initialized instance:

```
title: aString
 ^(super new)
  title: aString;
  yourself
```

Building the Descriptor System

To model the domain classes, the descriptor system class needs the following basic methods:

```
allTableNames
 ^#('CUSTOMERS' 'BOOKS' 'BOOKS_ON_ORDER')
```

```
constructAllClasses
^(super constructAllClasses)
 add: Customer;
 add: Book;
 yourself
```

To model the tables for this domain model, three methods are required:

```
tableForCUSTOMERS: aTable
 (aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey.
 (aTable createFieldNamed: 'first_name' type: (platform varChar: 50)).
 (aTable createFieldNamed: 'last_name' type: (platform varChar: 50)).

tableForBOOKS: aTable
(aTable createFieldNamed: 'title' type: (platform varChar: 100)).
(aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey

tableForBOOKS_ON_ORDER: aTable
| custKey bookKey |
custKey := aTable
     createFieldNamed: 'customer_id'
     type: (platform int4).
aTable
 addForeignKeyFrom: custKey
 to: ((self tableNamed: 'CUSTOMERS') fieldNamed: 'id').
bookKey := aTable createFieldNamed: 'book_id' type: (platform int4).
aTable
 addForeignKeyFrom: bookKey
 to: ((self tableNamed: 'BOOKS') fieldNamed: 'id').
```

The third method describes the `BOOKS_ON_ORDER` table. This is used to hold the links between books and customers. Each row represents one book on order for a single customer. It uses two foreign keys, one to the `CUSTOMERS` table and one to `BOOKS table`.

The descriptor system also needs two descriptor methods. This is the only place we actually specify the use of a `ManyToManyMapping`:

```
descriptorForCustomer: aDescriptor
 | customerTable |
 customerTable := self tableNamed: 'CUSTOMERS'.
 aDescriptor table: customerTable.
```

```
(aDescriptor newMapping: DirectMapping)
 from: #firstName
 to: (customerTable fieldNamed: 'first_name').
(aDescriptor newMapping: DirectMapping)
 from: #lastName
 to: (customerTable fieldNamed: 'last_name').
(aDescriptor newMapping: DirectMapping)
 from: #id
 to: (customerTable fieldNamed: 'id').
(aDescriptor newMapping: ManyToManyMapping)
 attributeName: #booksOnOrder;
 referenceClass: Book

descriptorForBook: aDescriptor
| table |
table := self tableNamed: 'BOOKS'.
aDescriptor table: table.
(aDescriptor newMapping: DirectMapping)
 from: #title
 to: (table fieldNamed: 'title').
(aDescriptor newMapping: DirectMapping)
 from: #id
 to: (table fieldNamed: 'id').
```

Note that `descriptorForCustomer:` does not actually specify the name of the association table. Glorp deduces this by itself, but it expects the name to follow a certain pattern. It is also possible to explicitly name the table.

To complete the descriptor system, we must also add two methods to set up the class models:

```
classModelForCustomer: aClassModel
 aClassModel newAttributeNamed: #id.
 aClassModel newAttributeNamed: #firstName.
 aClassModel newAttributeNamed: #lastName.

 classModelForBook: aClassModel
 aClassModel newAttributeNamed: #id.
 aClassModel newAttributeNamed: #title.
```

Here, note that the `booksOnOrder` variable in class `Customer` is not described as an attribute of the class model.

Saving and Retrieving Domain Objects

To save some sample information in the database:

```
session beginUnitOfWork.
books := #('Green Eggs and Ham' 'Ulysses' 'Daisy Miller')
    collect: [:each | Book title: each].
session registerAll: books.
session register: (Customer first: 'Franz' last: 'Kafka').
session register: (Customer first: 'Theodore' last: 'Dreiser').
session commitUnitOfWork
```

To read some books from the database:

```
session beginUnitOfWork.
eggs := session
  readOneOf: Book
  where: [:each | each title = 'Green Eggs and Ham'].
joyce := session
   readOneOf: Book
   where: [:each | each title = 'Ulysses'].
session commitUnitOfWork
```

To place orders for books:

```
session beginUnitOfWork.
franz := session
   readOneOf: Customer
   where: [:each | each firstName = 'Franz'].
franz
 addBook: eggs;
 addBook: joyce.
ted := session
   readOneOf: Customer
   where: [:each | each firstName = 'Theodore'].
ted addBook: eggs.
session commitUnitOfWork.
```

Retrieve a customer's book order(s):

```
session beginUnitOfWork.
franz := session
   readOneOf: Customer
   where: [:each | each firstName = 'Franz'].
```

```
franz books
session commitUnitOfWork
```

Find all customers that have a particular book or books on order:

```
session beginUnitOfWork.
eggs := session
   readManyOf: Customer
   where: [:each |
     each booksOnOrder anySatisfy: [:book | book title like: 'Green%']].
session commitUnitOfWork
```

In the `where:` block of this code example, it is also possible to use `anySatisfyJoin:` in place of `anySatisfy:`. The resulting SQL will be slightly different, including an INNER JOIN.

Using a Read-only Mapping

Generally, the mappings defined in a descriptor can be used for both reading and writing data in database tables. However, it is also possible to create read-only mappings. These are defined on a column-by-column basis.

Read-only mappings are useful in several ways. For instance, a database might be shared by several applications, and yours is only allowed read access to certain columns. Alternately, if one part of your application writes to a column, while another part should not modify it, you can use a read-only mapping.

For example, if your application does optimistic locking using timestamps, the timestamp is automatically written out without even being mapped in the descriptor. You would not want to override that with a mapping that would cause a timestamp value to be written, but it's possible that you want to see when the data was last written.

A read-only mapping is defined in a descriptor, e.g.:

```
firstNameMapping := (aDescriptor newMapping: DirectMapping)
 from: #firstName
 to: (customerTable fieldNamed: 'first_name').
 firstNameMapping readOnly: true.
```

The contents of this column will not be disturbed during a write to the database. If a descriptor contains several mappings, one of which is read-write, it is of course still possible to write an object to the database, creating a new row. In this situation, the contents of the column maked read-only will be null.

To prevent all writes, i.e., to exclude the creation of rows in the table, all mappings of an object should be marked read only. There is no API for this, but the convenience method `beReadOnly` can be used with a simple loop, e.g.:

```
aDescriptor mappingsDo: [:each | each beReadOnly].
```

Read-only mappings are also useful when tuning the performance of an application. Recall that only registered objects get written to the database when a unit of work is committed. reading an object has the function of registering it. Further, any objects that are "reachable" from a registered object are also considered to be registered.

This means that in order to commit all of the changes in a unit of work, the entire graph of reachable objects must be searched. There is some overhead in this operation, and naturally more when changing many objects at once.

For example, if you have a human resources application which works with `Employee` objects, and each of these references a complex `Office` object, then both employee and office objects would be searched for changes when you commit changes to an employee object.

If your application is intended only to modify employee objects, you could mark the relation to `Office` read-only, and the corresponding office objects would not be traversed when completing a unit of work.

Putting it All Together: How to Code as Little as Possible

Whether you code your application's descriptor system by hand, or let Glorp's metadata patterns (Active Record or Atlas) generate the code for you, or use a mix of both, you only need enough information to disambiguate the system: Glorp deduces the rest of the fields and link tables needed.

To illustrate, consider the simple case of database tables `DRIVERS` and `VEHICLES`, with a many-many relationship table showing which drivers have driven which vehicles, and which vehicles have been driven by which drivers. Let's see how much — and how little — we can tell Glorp to map these tables to classes `Driver` and `Vehicle`.

If there is only a single relation in the schema between `Driver` and `Vehicle` then we can simply tell `Driver's` class model that it has a relationship to many vehicles: the `manyToMany` mapping in `Driver's` descriptor model will be deduced. I.e.,

```
classModelForDriver: aClassModel
…
 aClassModel
  newAttributeNamed: #vehicles
  collectionOf: Vehicle.
```

Alternatively, we can tell `Driver's` class model nothing, but tell its descriptor model that there's a many-to-many mapping to `Vehicle`.

```
descriptorForDriver: aDescriptor
…
 aDescriptor manyToManyMapping
  attributeName: #vehicles;
  referenceClass: Vehicle.
```

That will have the same effect; no more is needed. However, if the `DRIVER` table had another relationship to `VEHICLE` — a direct `oneToMany` to one or more main vehicles, say, as well as the non-direct `manyToMany` relationship to all vehicles driven — we could disambiguate by telling the mapping to look for a link table for this relationship:

```
usesLinkTable: true;
```

If that was not sufficient (if there were several routes between the tables), we could explicitly tell the mapping the join from DRIVER to the link table, expecting Glorp could deduce the rest from that.

```
join:
 (Join
  from: (driverTable fieldNamed: 'ID')
  to: (driverVehicleTable fieldNamed: 'DRIVER_ID'));
```

If that was not sufficient, we could tell mapping the link field(s) to use when seeking the onward route from the link table to the relationship's target,

```
linkFields:
  (Array with:
   (driverVehicleTable fieldNamed: 'VEHICLE_ID'));
```

or explicitly tell it that relationship as what Glorp calls the reverse join.

```
reverseJoin:
  (Join
   from: (driverVehicleTable fieldNamed: 'VEHICLE_ID')
   to: (vehicleTable fieldNamed: 'ID')).
```

(As it goes from link table to target, not target to link table, reverseJoin: is the reverse of what the join from the target would be — not the perfect name, perhaps, but that's what Glorp calls it.) And of course, we can tell Glorp all this information, not just some minimum amount.

To summarize: how much to tell Glorp is a question of what kind of refactoring robustness you expect to need. Not mentioning the table fields at all is more robust to their changing, but less robust to the adding of new fields and constraints that could make minimalist definitions ambiguous. Decide based on how you expect the database schema to evolve — or on simpler criteria like how much typing you feel like doing.

Migrating a Schema

While developing your application, you may need to change the structure of your domain classes (e.g., adding or removing instance variables). In this situation, naturally you also need to modify the schema class to reflect these changes. If your application already has persistent data, the table structure in the database must be modified as well.

Rather than dropping and recreating the tables to propagate the changes in your domain classes, you can follow a procedure known as *migration* to let Glorp modify the corresponding database tables automatically, bringing them into accord with your new (modified) schema. In this way, you can preserve the contents of the database.

The procedure to migrate a schema and database is straight-forward: first, clone or subclass your schema class. Make your modifications to the new class. Finally, use the migration API to modify the structure of the database tables. For this final step, you need to load the `GlorpMigration` parcel.

Behind the scenes, Glorp analyzes your schema classes to generate a migration script, and uses it perform the database updates. It is possible to create a number of successive versions of your schema, and migrate between several versions all at once. By following a simple pattern in the construction of your descriptor system class (described below), the schema analyzer can produce a script that updates an arbirary number of versions in a single pass.

To illustrate, let's say you have a simple domain class `Customer`, that is represented in your schema class, e.g.:

```
tableForCUSTOMER: aTable
 (aTable
  createFieldNamed: 'customer_id'
  type: platform int4) bePrimaryKey.
 aTable
  createFieldNamed: 'first_name'
  type: (platform varChar: 64).
 aTable
  createFieldNamed: 'last_name'
  type: (platform varChar: 64)
```

In the course of developing your application, let's say that you want to add a number of other variables to class `Customer`. Once that is done, you need to update the schema class. Rather than simply modifying the `tableForCUSTOMER:` method, you can migrate your schema as follows:

1. Subclass your schema class, using a name that clearly indicates that it is a new version. For example, if your existing schema class is called `MyDescriptorSystem`, you could create a subclass called `MyDescriptorSystem01`.
2. Create a `tableForCUSTOMER:` method in `MyDescriptorSystem01` that overrides the behavior of the superclass, e.g.:

   ```
   tableForCUSTOMER: aTable
    "Add the customer address information to the table."
    super tableForCUSTOMER: aTable.
    aTable createFieldNamed: 'address1' type: (platform varChar: 64).
    aTable createFieldNamed: 'address2' type: (platform varChar: 64).
    aTable createFieldNamed: 'city' type: (platform varChar: 64).
    aTable createFieldNamed: 'state' type: (platform varChar: 2).
    aTable createFieldNamed: 'zip' type: (platform varChar: 9).
    aTable createFieldNamed: 'phone' type: (platform varChar: 20).
   ```

   Note that the superclass method is used to create the basic fields. You can add or remove tables by overriding the `allTableNames` method, and modifying it appropriately.
3. Create a class-side method called `migrationAncestor` in class `MyDescriptorSystem01` that returns the previous version of the schema class. E.g.:

   ```
   migrationAncestor
    ^MyDescriptorSystem
   ```

   This method is used to establish the ancestry of your schema classes. It is not necessary to define this method in the first (oldest) version of your schema, only in successive versions.
4. Migrate the schema using the following code:

   ```
   aLogin := (Login new)
       database: PostgreSQLPlatform new;
       username: 'userName';
   ```

```
    password: 'password';
    connectString: '127.0.0.1:5432_postgres'.
sessOld := MyDescriptorSystem new sessionForLogin: aLogin.
sessNew := MyDescriptorSystem01 new sessionForLogin: aLogin.
sessOld system migrateTo: sessNew system.
```

This example code will build a migration script and actually modify the database.

Note: When using foreign keys to link tables, some vendors require that the table exists before you can add a link field. Similarly, if you remove a table, any links to it must be removed first. The Glorp migration logic does not attempt to script these actions in a consistent sequence, so it is the developer's responsibility. The solution is to simply use separate migration steps to add a table, then add the link to it; or, to remove a link, and then the corresponding table.

Using Classes as Storage Managers

Some persistence frameworks employ a pattern whereby the domain classes also serve as storage managers. In this design, class-side methods are added that explicitly read and write instances to the database.

When using this pattern, persistence is neither automatic nor transparent, because each object needs to be explicitly read from and written to the database. This design typically results in a proliferation of specialized query methods added to each domain class.

Glorp can be used to persist domain objects using this strategy, but it is discouraged because of a number of well-known design problems. Instead, Glorp provides an alternative approach which, when understood, can simplify the design of your application. You can either use Glorp to model your domain classes, or use the Active Record pattern to automate the modeling.

Chapter

6

Units of Work

Topics

Once your domain classes have been modeled using a descriptor system, and your application has established a connection with the database server, it can use a session object to issue commands. Typically, you must create tables before you can save and retrieve your application's domain objects.

Within each session, all manipulation of persistent data is organized using *units of work*. These are a higher level of abstraction than simple database transactions, as they allow you to perform true object-level transactions. That is, if your domain model includes complex classes such as `Set` or `Dictionary` that span several different tables, you can commit or rollback changes that reflect the object-oriented structure of your application's domain model.

Managing Tables

Your application may use pre-existing tables in a database, tables that were created with some other application, or it may create and drop the tables itself.

Glorp provides a simple API for manipulating database tables, in the event that your application needs to explicitly manage them.

Creating Tables

Glorp may be used to create database tables programmatically.

Since your application typically includes a descriptor system class that defines all the tables used by your application, it is possible to create them with a single message:

```
aSession createTables
```

This creates the tables for the descriptor system associated with `aSession`, as well as their associated sequences. Note that if the tables already exist, an exception will be raised.

Alternately, you may use:

```
aSession recreateTables
```

This drops and re-creates the tables for the descriptor system associated with `aSession`, as well as their associated sequences. Any errors will be trapped.

It is also possible to create and drop tables using a database accessor alone, by explicitly assembling a `DatabaseTable` object. Generally, this approach is discouraged in favor of using the metadata in the descriptor system, but there may be occasions when you wish to do this.

For example, consider a table called `Customer` that has two columns: a `cust_id` column containing an integer, and a `cust_name` column containing up to 255 characters. Assume that `cust_id` is the key for each customer object. This table could be created using the following SQL:

```
CREATE TABLE customer(
 cust_id integer CONSTRAINT customer_PK PRIMARY KEY,
 cust_name varchar(255));
```

To express the definition of this table using Glorp, assemble an instance of `DatabaseTable` as follows:

```
table := DatabaseTable named: 'customer'.
keyField := table
     createFieldNamed: 'cust_id'
     type: accessor platform int4.
table addAsPrimaryKeyField: keyField.
table
 createFieldNamed: 'cust_name'
 type: (accessor platform varChar: 255).
```

To actually create the table, pass this object to a `DatabaseAccessor`:

```
accessor
 createTable: table
 ifError: [:ex | Transcript show: ex messageText].
```

This message will autocommit. Note that if you use sequences (i.e., primary keys that auto-increment), you may need to create them explicitly. The message `createTables` does this for you, while `createTable:ifError:` does not.

Deleting Tables

To delete all database tables associated with your domain model, evaluate the following:

```
aSession dropTables
```

It is also possible to create and drop individual tables using a database accessor object, either by specifying a `DatabaseTable` object or the name of the table as a `String`. This approach is discouraged in favor of using dropTables, but there may be occasions when you wish to do this.

For example, to drop a table using a DatabaseTable object:

```
accessor
 dropTable: table
 ifAbsent: [:ex | Transcript show: ex messageText].
```

This message will autocommit.

To drop the table using only its name:

```
accessor dropTableNamed: aString.
```

Or, to do the same with simple error handling:

```
accessor dropTableNamed: table ifAbsent: [].
```

Managing Units of Work

Any time you wish to insert, remove, or update objects in the database, you should wrap these operations in a *unit of work*.

A Glorp unit of work follows the basic pattern of a database transaction, but it is actually a higher-level operation. When a unit of work is committed, it gathers together the modified objects associated with that unit and writes them to the database.

The difference between a unit of work and the underlying database transaction is that the former can also roll back changes to Smalltalk objects in memory. I.e., your application can register an object within a unit of work, manipulate the object, and in the event of an exception, roll back any changes to the object. Glorp automatically wraps each unit of work in a transaction, so there is seldom a need to manipulate transactions explicitly. Instead, you should begin and commit units of work.

Usage

Glorp provides the following protocol (instance behavior in class GlorpSession) to initiate, commit, and roll back units of work:

beginUnitOfWork

Explicitly begin a unit of work. Each session object can only have a single, concurrent unit of work, i.e., they cannot be nested.

commitUnitOfWork

Commit the current unit of work, writing any modified objects that have been registered with the current session to the database. Once the unit of work has been committed, flush the list of registered objects.

rollbackUnitOfWork

Rollback the current unit of work, restoring any registered objects to their state when first registered with the session.

inUnitOfWorkDo: aBlock

Evaluate aBlock within a unit of work. If the block evaluates without raising any exceptions, commit the unit of work. If an exception is raised, roll back the unit of work.

commitUnitOfWorkAndContinue

Commit the current unit of work, but then keep going with the same set of registered objects, with their state updated to reflect current values.

When performing units of work, Glorp also uses an *object cache*. This is important for maintaining object identity and object-level transactions. Any objects that are registered with Glorp enter the cache, and when a unit of work is completed, the contents of the cache are used to determine which objects need to be inserted or updated in the database. Similarly, if a unit of work is rolled back, the cache is used to restore objects to their previous state.

As a general rule of thumb, consider coding with units of work by default, using optimistic locking. If you encounter problems relating to inconsistent data or concurrency, it may be necessary to make more explicit use of transactions. Generally, though, concurrency conflicts are rare, though of course it depends on the application area.

Inserting Objects in the Database

When an application using Glorp wishes to add objects to a database, it does so by *registering* them with the current session object. This must take place inside a unit of work, because you are modifying the database.

For example, with a properly initialized session object, you may add objects to the database using `register:`. E.g.:

```
aSession beginUnitOfWork.
aCustomer := Customer
   firstName: 'Richard'
   lastName: 'Jones'.
aSession register: aCalendarEvent.
aSession commitUnitOfWork;
```

Here, the effect of registering an object is to include it in the current unit of work. When the unit of work is committed, any registered (and modified) objects are written to the database.

You can also roll back any changes using `rollbackUnitOfWork`. Note, however, that changes made to an object before it is registered with a session cannot be rolled back in the event that the unit of work is aborted.

Several shorter versions of this pattern are also available, e.g.:

```
aSession inUnitOfWorkDo: [aSession register: aCustomer].
```

Or:

```
aSession modify: aCustomer in: [].
```

The correct schema for translating the object into relational form is found via its class. The object is then added to the appropriate table(s) as a new row. Typically, your domain classes will have an instance variable named `id` to hold the object's primary key. If you register a domain object that has a `nil` value for `id`, and if it is declared to be a `sequence`, Glorp automatically fetches and assigns the next sequential value to `id`.

If you choose to set id explicitly, Glorp will still try to add it to the table. Note that if the object was read from the database and then modified within the same unit of work, Glorp will update the same row of the database when the unit of work is committed.

Grouped Inserts

When inserting a collection of objects in the database, Glorp automatically tries to optimize the underlying SQL operation. Some vendors (e.g., Oracle) support a feature called "Array binding" that makes it possible to insert a collection of objects in a single operation. If this feature is available, Glorp uses it.

Even if Array-binding is not supported by the database, Glorp still tries to group a series of inserts into a single operation. Since each operation involves a separate round-trip to the database server, this can eliminate some potential network latency.

The number of statements that may be grouped are limited, and the statements are not grouped if your application needs to get results back from each operation. In the latter case, a sequence is typically used to pre-allocate a primary key for each row being inserted.

Reading Objects from the Database

Glorp provides two different APIs for reading objects from a database. For simple queries, you can send messages directly to the session object, while for more complex ones, you can create specialized, reusable Query objects.

Let’s begin with the simple case. With a properly initialized session object, you may read objects from the database using readOneOf: or read:, where the argument is simply the class of the objects that you wish to read.

For example, to read a single object, use the following:

```
passenger := aSession readOneOf: Passenger.
```

This method returns the first object that satisfies the selection criteria.

To read more than one object:

```
people := aSession read: Person.
```

This returns a collection of `Person` objects. In effect, `read:` is the shorter (and thus preferred) form of the functionally equivalent `readManyOf:`.

All reads and queries performed using Glorp also involve an *object cache*. This maintains the identity of Smalltalk objects and their database representations. That is, if your application fetches the same database row in two different sections of the code, the same Smalltalk object is returned. Identity is preserved by all Glorp APIs.

Reading Objects with Simple Criteria

To read a collection of objects and restrict the number of results to a specific number:

```
people := aSession read: Person limit: 50.
```

This returns a collection of the first 50 objects that satisfy the selection criteria. Note: to read objects starting with object N + 1 that matches, you need to build an explicit `Query` object.

To order the results:

```
people := aSession read: Person orderBy: #lastName.
```

Or:

```
people := aSession
  read: Person
  orderBy: [:each | each lastName].
```

Or:

```
people := aSession
  read: Book
  orderBy: [:each | each price descending].
```

The argument to `orderBy:` may be either a `Symbol` or a `Block`. The former to identify an instance variable (i.e., a column) used for sorting the results, while the latter may contain a more complicated expression.

Note, however, that there are some important restrictions on the code that may appear in the query block passed to `orderBy:`.

Removing Objects from the Database

With a properly initialized session object, you may remove objects from the database using `delete:`.

```
aSession
 beginUnitOfWork;
 delete: aCalendarEvent;
 commitUnitOfWork;
```

Here, the explicit handling of a unit of work is in fact not strictly necessary, as a delete operation will start and commit a unit of work if there isn't one already started.

However, if you want to delete several objects atomically, then the delete operations must occur within a unit of work, as shown above.

Glorp also provides the convenience method `deleteAll:`, for removing a collection of objects. E.g.:

```
people := aSession
 read: Person
 where: [:each | each firstName in: #('Larry' 'Moe' 'Curly')].
aSession deleteAll: people.
```

Updating Objects in the Database

With a properly initialized session object, you may update objects in the database simply by reading and modifying them within a unit of work.

For example, the following code retrieves a person, changes the first name, and then writes the changes back to the database:

```
session beginUnitOfWork.
aPerson := session
   readOneOf: Person
   where: [:each | each firstName = 'Joe'].
```

```
aPerson firstName: 'Frank'.
session commitUnitOfWork
```

Each unit of work keeps track of changes made to any objects read from the database within the same session. When the unit of work is committed, it writes all the changed objects back to the database.

Sending `commitUnitOfWork` clears the list of objects being tracked by the session. To retain this list of objects, use `commitUnitOfWorkAndContinue` or `saveAndContinue` instead of `commitUnitOfWork`.

We can also rollback a unit of work:

```
session beginUnitOfWork.
aPerson := session
    readOneOf: Person
    where: [:each | each firstName = 'Joe'].
aPerson firstName: 'Frank'.
session rollbackUnitOfWork.
```

Note that when you rollback a unit of work, all the objects in the unit of work are also reverted to their previous state. That is, all the changes made to the objects themselves are undone. In the example shown above, the rollback restores the name `'Joe'` to the object `aPerson`.

As a shorthand, we can update objects in the database using `inUnitOfWorkDo:`, e.g.:

```
session inUnitOfWorkDo:
 [aPerson := session
     readOneOf: Person
     where: [:each | each firstName = 'Joe'].
 aPerson firstName: 'Robert'].
```

Here, the unit of work is automatically committed if no exception is raised when executing the block. If an exception is raised, the unit of work is rolled back.

Managing Transactions

A transaction is a technique for manipulating shared data in a way that is consistent in the view of other clients, while maintaining the integrity of the data store. Most modern relational databases (including all vendors supported by Glorp) are transactional databases.

In discussions of relational databases, transactions are sometimes referred to as "units of work" though in the context of using Glorp, the two phrases are not necessarily identical. With Glorp, a unit of work has slightly different semantics than a basic database transaction.

Glorp provides protocol (via a `GlorpSession` object) to manipulate both database transactions (which are lower level) and Glorp's own units of work.

Note: as a general rule it is best to structure your application using units of work, rather than low-level transactions.

Usage

Using a `DatabaseAccessor` object, you can begin, commit, and rollback database transactions. E.g.:

```
accessor beginTransaction.
...
accessor commitTransaction.
...
accessor rollbackTransaction.
```

As a rule, though, your application should use units of work rather than the transaction API.

Chapter

7

Queries

Topics

Glorp provides two different APIs for making queries against the database. For simple queries, you can send messages directly to the session object (e.g., `aSession read: Person`). Alternately, you can create specialized, reusable `Query` objects.

Query objects are generally used when there is a need for more complex selection criteria, or to defer or separate the composition from the execution of the query.

The execution of a query object may involve several sub-queries, either as a compound that gets combined via union or intersections, or it may involve a sequence of calls to the database server, all of which are handled by Glorp. To facilitate this, the framework provides a variety of specialized query objects that may be used for composing more elaborate database queries.

Building Queries

A query object is assembled by your application and then subsequently executed by Glorp. Class Query and its subclasses are used to represent the various types of queries, and while Query is an abstract class, it may also be directly instantiated.

For example, a query is assembled and then executed like this:

```
userQuery := Query read: Person.
userQuery
 orderBy: [:each | each name descending].
session execute: userQuery.
```

Here, the userQuery object may be used repeatedly.

Query objects can be created for reading a single object (using readOneOf:), or many objects (using read: or readManyOf:, which are synonymous). Queries may be constrained using orderBy:, which expects a Symbol, an Array, or a query block. For example:

```
"Order the query results by the #name column"
userQuery orderBy: #name.
...
"Order the query results by the #address and #street columns"
userQuery orderBy: #(#address #street).
...
"Order the query results by the #id column"
userQuery orderBy: [:each | each id].
```

To assemble more complex queries, multiple orderBy: messages can be sent to a single query object, e.g.:

```
userQuery := Query read: Address.
userQuery
 orderBy: [:each | each street];
 orderBy: [:each | each number].
session execute: userQuery.
```

Using Query Blocks

Glorp uses a *query block* to generate SELECT statements that are executed by the database server. Sometimes referred to as a *where clause*, this block is used to specify selection criteria, e.g.:

```
aSession
 readOneOf: Person
 where: [:each | each firstName = 'Sam'].
```

While it outwardly resembles a standard Smalltalk `BlockClosure`, a query block has special semantics which application developers need to observe for proper use.

To perform the query, the code in the where clause is parsed and translated into SQL (i.e., the WHERE portion of a SELECT statement), rather than being executed as a Smalltalk `BlockClosure`. In effect, query blocks permit a subset of Smalltalk syntax, and are subject to two important restrictions.

The first restriction on a query block concerns the messages that may be sent to the block argument. These must be:

- The names of instance variables in the object we are trying to read, and these variables must be mapped to fields in the database.
- Binary operators: `=`, `<>`, `~=`
- `notNIL`, `isNIL`
- `in:`

The first two of these rules are illustrated in the code example shown above. There, the block argument `each` is sent `firstName` to access the corresponding column, and the result is compared using `=`.

The messages `=` and `<>` may be used to read objects using selection critera that involve other Smalltalk objects. Note that the messages `~=` and `<>` are equivalent: both generate the SQL `<>` operator. To illustrate, the following code reads the first four `Person` objects who are not `'Sam'`:

```
sam := aSession
```

```
  readOneOf: Person
  where: [:each | each firstName = 'Sam'].
result := aSession
  readManyOf: Person
  where: [:each | each <> sam]
  limit: 4.
```

In the second query, the block `[:each | each <> sam]` makes a comparison based on the primary key of the object `sam`.

Note that the class of the object being read doesn't actually need to define accessors for the instance variables, because the query block is only being used to generate SQL, and the names of the instance variables are mapped to column names. Thus, in the example code above, class `Person` doesn't need to define an accessor for `firstName`.

The `notNIL` and `isNIL` messages generate `IS NULL` and `IS NOT NULL` tests on the primary key of the table. These have the effect of selecting either the entire table or no elements, and are obviously of limited utility.

The message `in:` is equivalent to the `IN` operator, and may be used to select objects that have a column matching a small collection of known values. For example, to find all the `Address` objects located on a set of known streets:

```
addresses := aSession
   readManyOf: Address
   where: [:each | each street in: #('Main' 'Market' 'Park')].
```

The second restriction on a query block concerns the messages that may be sent to instance variables inside the block. These messages include:

- Binary operators: `<`, `>`, `=`, `~=`, `>=`, `<=`, `<>`
- `like:`
- `isNIL`, `notNIL`
- `AND:`, `&`, `OR:`, `|`
- Methods in the API protocol of class `ObjectExpression`

The binary operators may be used to compare database columns with constants or simple Smalltalk objects:

```
result := aSession
```

```
  readManyOf: Person
  where: [:each | each id >= 1].
numbers := #(1 2 3).
result := aSession
  readManyOf: Person
  where: [:each | each id >= numbers first].
```

The like: message generates a SELECT statement using the LIKE operator. For example, to read all Person objects whose first name starts with the letter "S":

```
result := aSession
  readManyOf: Person
  where: [:each | each firstName like: 'S%' ].
```

Here, the "%" character functions as a wildcard, matching any sequence of characters. Note that some databases perform case-sensitive comparison (e.g., PostgreSQL), while others do not (e.g., MySQL).

To generate the NOT LIKE clause:

```
result := aSession
  readManyOf: Person
  where: [:each | (each firstName like: 'S%') not].
```

The messages isNIL and notNIL are used to find an object in which a field is NULL or not. For example, to find all people in the database whose first name is not NULL:

```
result := aSession
  readManyOf: Person
  where: [:each | each firstName notNIL].
```

To perform logical AND and OR operations, query blocks should use the binary selectors AND:, &, and OR:, |. Note that while Smalltalk ordinarily allows lower-case and: and or: with block arguments, Glorp requires the upper-case AND: and OR:.

Logical operators can be combined with negation, as follows:

```
people := aSession
  readManyOf: Person
  where: [:each | (each firstName notNIL) &
     (each lastName like: 'Smit%') not].
```

From VisualWorks 7.10, a Glorp WHERE clause can now include arithmetic of Date objects. You can use both constant and field expressions, or dates added to or subtracted from integers, where the integer expression represents a number of days.

For example:

```
session
 read: Employee
 where: [:each | each startDate + 1 > Date today].
```

or you can use field expressions, like:

```
session
 read: Employee
 where: [:each | each startDate + each orientationDays < Date today].
```

Here, orientationDays is integer-valued.

You can also use duration values, e.g.:

```
session
 read: Employee
 where: [:each | each startDate + 5 days >= Date today].
```

Note: The DB2 platform does not allow integers, so you should use duration type expressions. We have therefore added Duration arithmetic functions to DB2Platform >> initializeFunctions.

For additional functionality available in query blocks, browse the api protocol of class Glorp.ObjectExpression.

Finally, you should be aware that since query blocks are used to generate SQL rather than being executed, using the Debugger to single step or set breakpoints inside the query block may produce unexpected results.

Reading Objects in a Particular Order

Database queries may be performed in a particular order. This has the effect of asking the database server to sort the results, which may, under certain circumstances, be more efficient than sorting them in your application.

For example, to fetch a collection of `User` objects and sort them by name:

```
aQuery := (Query readManyOf: User)
  orderBy: #firstName;
  yourself.
session execute: aQuery.
```

To build more complex sorting rules, multiple `orderBy:`messages may also be sent to a single query object, i.e., to order by A, then B, then C, etc.:

```
aQuery := (Query readManyOf: User)
  orderBy: [:each | each name descending];
  orderBy: #joined;
  yourself.
session execute: aQuery.
```

The argument to `orderBy:` is either a `Symbol` or a query block.

Using Detached Objects

Once an object is read from the database, your application may hold a reference to it even after the session object or database accessor has been closed. In this situation, the object is said to be *detached* from the database. If Glorp subsequently attempts to dereference a proxy in a detached object, an exception is raised. Note, however, that if the underlying accessor object has been closed but the session remains open, the proxy may still be used.

To reattach the object to the caching mechanism, use `register:`, e.g.:

```
aSession register: aCustomer.
```

Locking Objects in the Database

As a general rule of thumb, applications that allow multiple users to modify the same database objects should use some form of locking. Locking is important for maintaining the integrity of data when several users could modify the same persistent object. While this is a relatively rare occurance, it needs to be managed properly.

Glorp provides support for so-called optimistic locking. This strategy is favored because under normal circumstances it is non-intrusive. By following this pattern, applications can allow multiple users to read the same object, and handle exceptions only when one user attempts to update a shared object out of sequence with other users.

To lock objects in a database table, one column is used as a locking key. Typically, this is a `Number` or a `Timestamp`. Alternately, every column in the object may be set as a locking key. This option is slower, but useful if it is not possible to modify the database schema to include such a field.

By sending `beLockKey` to a column (actually, to a `DatabaseField` object), it is marked as a locking key. This is typically done in a `tableForNAME:` method in a descriptor system class. For example:

```
(aTable
 createFieldNamed: 'VERSION'
 type: (platform versionFieldFor: platform int4)) beLockKey.
```

When a database column is marked for locking and the application updates it in the database, Glorp checks the copy held in memory against the copy in the database. If the field marked as a locking key has been modified, a `GlorpWriteFailure` exception is raised and the application may take appropriate action.

If your application receives a `GlorpWriteFailure` when committing a unit of work, there are several options.

You may ignore the locking error and just write the object anyway by sending `resume: true` in the exception handler. E.g.:

```
[self updateSomeObject]
```

```
on: GlorpWriteFailure
do: [:ex | ex resume: true].
```

Under most circumstances, this approach is inadvisable. After all, the point of locking is to protect against exactly this circumstance.

Alternately, you may let the transaction fail, selectively refresh the objects that were modified by another thread, re-apply the changes, and try the unit of work again. It may be necessary to perform a larger-scale refresh or notify the user that a conflict was encountered, giving the option to refresh and try again.

The `GlorpWriteFailure` exception holds the object or objects that caused the failure. These may be obtained by simply sending the message `object` to the exception object. Note that if the exception was raised while trying to write multiple rows in a single database command, it may not be obvious exactly which one was the problem object.

Querying with Collections

When reading domain objects that contain collections, your application can make specialized queries across these relationships.

For example, in an application that tracks book orders, let's say that each `Customer` object includes a collection of `Book` objects, stored in an instance variable named `booksOnOrder`. To find all the customers who have a particular book on order, you can query against this nested collection, as follows:

```
session beginUnitOfWork.
bookName := 'Cat in the Hat'.
customers := session
  readManyOf: Customer
  where: [:each |
    each booksOnOrder
      anySatisfy: [:book | book title like: bookName]].
session commitUnitOfWork
```

Here, the result is a collection of `Customer` objects that have ordered the specified book.

In addition to anySatisfy:, you can also use anySatisfyJoin:, or noneSatisfy:. Functionally, anySatisfy: and anySatisfyJoin: are similar, with the difference being in the SQL code generated for the SELECT statement. For the former, WHERE EXISTS is generated with a nested SELECT, while in the latter SELECT DISTINCT is used with an INNER JOIN.

Retrieving and Grouping Specific Values

If specific values within mapped objects are wanted, not whole instances, it can be more efficient to retrieve just these desired values from the database. For example, a query can be sent the message retrieve: with a symbol, e.g.:

```
query retrieve: #name.
```

or a block, e.g.:

```
query retrieve: [:each | each name distinct].
```

A readOneOf: query will return an Array with the requested value. A read: query will return an Array of the requested values. If more than one value is requested, an Array (readOneOf:) or an Array of Arrays (read:) is returned. For example,

```
query := Query read: StoreBundle.
query retrieve: [:each | each name].
query retrieve: [:each | each timestamp].
session execute: query
```

might return:

```
#(#('Glorp' 13 June 2012 18:04:43)
 #('Glorp' 15 June 2012 13:27:05)
 #('StoreBase' 15 June 2012 13:29:22)
 #('StoreBase' 22 June 2012 08:07:51))
```

In queries where some column values in the returned rows may have duplicates, retreving can be combined with grouping, so a single row is returned for all rows with the same values for a specified column, along with summary data for other retrieved values.

For example, in the query:

```
query := Query read: StoreBundle.
query retrieve: #name.
query retrieveMax: #timestamp.
query groupBy: #name.
query orderBy: #name.
session execute: query
```

the data in the array of arrays above would appear as:

```
#(#('Glorp' 15 June 2012 13:27:05)
#('StoreBase' 22 June 2012 08:07:51))
```

(Note that the third line

```
query retrieveMax: #timestamp.
```

in the above query is a short form of:

```
query retrieve: [:each | each timestamp max].
```

Other such convenience methods are `retrieveMin:` and `retrieveSum:`.)

We can group on several columns.

```
query := Query read: GlorpBook.
query retrieve: [:each | each copiesInStock sum].
query retrieve: [:each | each countStar].
query groupBy: [:each | each title].
query groupBy: [:each | each version].
query orderBy: [:each | each title].
session execute: query.
```

As this example shows, it is not necessary to retrieve the grouped-by columns. However, a result of:

```
#(#(5 2) #(7 3))
```

may be less useful than:

```
#(#('Animal Farm' '1st Edition' 5 2) #('1984' '2nd Edition' 7 3))
```

obtained by including:

```
query retrieve: #title.
query retrieve: #version.
```

Retrieving and grouping are usually used in queries, rarely in descriptors, unlike ordering — see references to `orderBy:` in preceding topics.

Note: It is of course easy to write a `groupBy:` query that would return conflicting values for one of the returned fields. When such a miswritten query is run, it raises an error demanding omission of that field or aggregation of its values. Aggregation can be done via `sum`, `max`, `min`, `countStar` (counts rows), `count` (counts values in specific field), `average`, `aggregate:as:` (a more general syntax) or similar. Examples of use of these are in the `GlorpTest` parcel.

Retrieval expressions are not limited to the above list; they can be computed, e.g.:

```
query retrieve: [:each | each quantity * 3]
```

Anything done by grouping and retrieving can of course be done by reading the instances and then processing them in the image, but the speed and memory costs may differ greatly. This API lets you perform on the server computations for which relational databases are optimised.

Recursive Queries

Multiple round trips to the server, each one retrieving some data needed to define the next query, can be a weak link in a high performance database application. Glorp can leverage the growing power of modern database servers to retrieve recursive data in a single round trip.

Recursive SQL is supported on more recent versions of several databases, including the following and all later versions: PostgreSQL version 8.4, Oracle version 11.2, SQLServer version 10.5 (version

2008 R2D) and SQLite version 3.8.3. DB2 can recurse in version 7 and after, but does not support outer joins in recursive statements.

Recursive SQL allows a single query in a single round trip to put the rows returned by an initial subquery into a temporary "common table expression", and then write successive rows to it using a recursive subquery whose input data is the last set of rows found. The temporary table is then joined to other tables in a final query whose rows are the ones actually returned.

As an example, a recursive SQL query might look like this:

```
"First we define the common table expression."
WITH RECURSIVE ancestor(primaryKey, trace) AS
( "529 is id of some bundle whose ancestors we will read"
 (SELECT primarykey, trace
  FROM tw_bundle
  WHERE primarykey = 529)
UNION ALL "unites initial subquery to recursive subquery"
 (SELECT a1.primaryKey, a1.trace
  FROM tw_bundle a1, ancestor a2
  WHERE  a1.primarykey = a2.trace)
)  "Finally, select the rows to return from common table and others"
 SELECT t1.* from tw_bundle t1, ancestor t2
 WHERE t1.primaryKey = t2.primaryKey
```

This query returns the ancestors of a store bundle with id `529`.

In Glorp, a `RecursiveQuery` can be created from the initial and final subqueries, plus the information of what mapping is to be iterated over and what is to be retrieved to the temporary table in each iteration. For example, the query shown above can be run from the Store Workbook by executing:

```
| query descendantId |
descendantId := 529.  "id of bundle whose ancestors are to be found"
query := (Query  "subquery for the initial row(s)"
 read: StoreBundle
 where: [:each | each primaryKey = descendantId])
retrieveAll:  "the initial subquery must retrieve to define the"
 (Array  "columns of the common table expression"
  with: [:each | each primaryKey]
  with: [:each | each trace primaryKey])
thenFollow: #trace  "the mapping whose closure is wanted"
```

```
recursivelyRetrievingAll:  "these retrievals populate the recursion"
 (Array  "the #recurse mapping is temporarily defined"
  with: [:each | each recurse primaryKey]
  with: [:each | each recurse trace primaryKey])
intersect:
 (Query  "the #recursed mapping is temporarily defined"
  read: StoreBundle
  where: [:each | each primaryKey = each recursed primaryKey]).
session execute: query.
```

A recursive query defines one temporary table and three temporary mappings in a clone of the descriptor system it is using. The table is the one that will appear in the `WITH` clause. The mappings are:

- The `recurse` mapping connects the temporary table back to the initial expression. It is typically used in the second `retrieving:` or `retrievingAll:` blocks to connect the table to the values. For example, the block:

  ```
  retrieving: [:each | each recurse children id]
  ```

  tells the recursive subquery to get the `ids` of the children of the rows it found in the prior recursive step, where `children` is a mapping in the `DescriptorSystem`.
- The `recursed` mapping is the inverse, connecting the base object back to the temporary table. It is typically used in an ordering clause of the final query. For example, in:

  ```
  (Query read: GlorpCity where: [:each | each name = 'London'])
   retrievingAll: (Array
     with: [:each | primaryKey]
     with: [:each | name]
     with: [:each | 0 alias: #depth]) "not a column, so must alias"
   thenFollow: #nextCity
   retrievingAll: (Array
     with: [:each | each recurse primaryKey]
     with: [:each | each recurse name]
     with: [:each | each depth + 1 cast: Integer])
   intersect: ((Query read: GlorpCity)
       orderBy: [:each | each recursed depth]).
  ```

  the rows that the query returns — and therefore the objects that Glorp returns — will be ordered by the depth of recursion.

(The aliasing of `depth` in the first `retrievingAll:` is essential: it is not a column, so has no name, so users must give an alias to refer to it in the `orderBy:` clause. The casting of the `depth` expression to `Integer` in the second `retrievingAll:` is merely precautionary — most platforms will not need that.)

- The third mapping is named by attaching the suffix `'Recursively'` to the name of the mapping being followed. Thus in the examples above, it would be called `nextCityRecursively`. It connects the `Query`'s base object to itself and returns the closure of the followed mapping. Thus in the example above, if `nextCity` returned `'Oxford'` (as the next city to `'London'` on some route that is the data), then `nextCityRecursively` would return `'Oxford'` and Oxford's `nextCity` and that city's `nextCity` and so on.

The `recursed` mapping can be used in the final query's `WHERE` clause, to define how the final query relates to the recursion table. If it is absent (as in the examples above), the `RecursiveQuery` will add the join (the system must have one since the initial step joins the base table(s) to the temporary recursion table using the retrieved field(s)). Thus:

```
...
intersect:
 (Query
  read: GlorpCity
  where: [:each | each primaryKey = each recursed primaryKey]).
```

adds explicitly the join that would be added automatically if the where clause was not provided.

As a final caveat, it is worth bearing in mind that during the process of development and debugging there is a danger of any recusive algorithm going haywire, leading to a runaway query which might offend your DBA. It is best to develop your recursive query on a private server using a small database, initially. See the `GlorpTests` package for hints on how to limit recursion depth, or output size.

Chapter

8

Working with Complex Objects

Topics

Glorp uses classes to model database tables, with each instance variable in a domain class being mapped to a single database column. This design works well for domain classes whose instance variables contain simple objects such as numbers, strings, dates, etc.

For classes that contain more complex objects, such as collections or custom domain classes, Glorp uses separate tables that are related using foreign keys. For example, if class `BankAccount` has an instance variable that holds an instance of class `Customer`, two different tables are used in the database, and the `BankAccount` class is considered "complex".

When using any kind of complex object, suitable mappings must be chosen to model the relationship.

For certain types of complex domain objects such as collections, Glorp uses *proxies* to load them from the database on demand. That is, for an initial query, only part of a domain object may be loaded, with proxy objects in place of its instance variables.* If and when an instance variable is referenced, the framework automatically resolves the proxy and loads the referenced object. Proxies are managed internally by Glorp, and your application generally does not need to handle them. It is important, though, to have some basic understanding of their semantics.

* For a discussion of the so-called "Lazy Load pattern", see p. 200-214 in Fowler [2003].

Cascading

Often, your domain model may include nested objects, e.g.:

```
aCustomer := Customer firstName: 'Franz' lastName: 'Kafka'.
aBook := Book title: 'Green Eggs and Ham'.
aCustomer addBook: aBook.
```

In this example, when your application saves a `Customer` object in the database, generally you want any nested objects to be written as well. This behavior is known as *cascading*. By default, Glorp cascades all writes to the database.

Thus, the following line of code will write both the `Customer` and any nested `Book` objects:

```
session inUnitOfWorkDo: [session register: aCustomer].
```

Similarly, any time your application updates an object, a cascaded write occurs when the enclosing unit of work is complete. For example:

```
session beginUnitOfWork.
franz := session
  readOneOf: Customer
  where: [:each | each firstName = 'Franz'].
aBook := Book title: 'Green Eggs and Ham'.
franz addBook: aBook.
session commitUnitOfWork.
```

Here, we are updating an existing `Customer` object to include a nested `Book` object, and it gets saved to the database along with the update.

While inserting or updating an object performs a cascaded write, deleting an object will not. For instance, the following code deletes the specified `Customer` object, but not the books that it references (after all, they may be referenced by other customers):

```
session
 inUnitOfWorkDo:
  [franz := session
    readOneOf: Customer
    where: [:each | each firstName = 'Franz'].
  session delete: franz].
```

To remove any nested objects, you must delete them explicitly.

Alternately, when defining the mapping between `Customer` and `Book`, you can use the message `beExclusive` to instruct Glorp to perform a cascaded delete. E.g.:

```
(aDescriptor newMapping: ManyToManyMapping)
 attributeName: #booksOnOrder;
 referenceClass: Book;
 beExclusive.
```

Refreshing Objects from the Database

When a database is accessed concurrently by different threads of execution, or shared by several distinct applications, it is possible for the data in the object cache to be different than that in the database. While your application can use optimistic locking to maintain the correct sequence of writes, there may be cases in which you know that the data in memory needs to be refreshed from the database. In these situations, you can use `refresh:` to explicitly update objects in memory:

```
anOffice := aSession
   readOneOf: Office
   where: [:each | each name = 'Headquarters'].

"later..."
aSession refresh: anOffice.
```

Direct Access or Accessor Methods

By defalt, Glorp makes direct reference to instance variables in your domain classes. That is, it does not use accessor methods. Generally, this is fine, but in the event that you want all references to be made via accessor methods, you can set up the class model as follows:

```
classModelForPerson: aClassModel
 aClassModel newAttributeNamed: #lastName.
 (aClassModel newAttributeNamed: #firstName)
  useDirectAccess: false.
 aClassModel newAttributeNamed: #id.
```

The method `useDirectAccess:` should be invoked for each instance variable to be referenced via accessors.

You can also change the default for an entire descriptor system, e.g.:

```
myDescriptor useDirectAccessForMapping: false.
```

To make the change at the level of the framework itself (i.e., to use accessor methods for all descriptors), you can override the method`useDirectAccessForMapping`, in class `GlorpTutorialDescriptor` e.g.:

```
useDirectAccessForMapping
 ^false
```

Specifying the Type of a Collection

When reading domain objects that contain collections, these are stored as instances of `OrderedCollection` by default. E.g.:

```
people := aSession read: Person.
firstAddress := people first emailAddresses first.
```

Here, the instance variable `emailAddresses` holds an `OrderedCollection`. If you wish to use a different collection class by default, you can specify it in the method that defines the class model, i.e., in the descriptor system class. For example:

```
classModelForPerson: aClassModel
 aClassModel newAttributeNamed: #id.
 aClassModel newAttributeNamed: #firstName.
 aClassModel newAttributeNamed: #lastName.
 aClassModel newAttributeNamed: #emailAddresses collection: SortedCollection of:
 EmailAddress.
```

When reading instances of `Person`, the instance variable `emailAddresses` will hold a `SortedCollection`. (Note: the domain class `EmailAddress` needs to implement the correct methods for sorting).

In this fashion, you can specify various types of collections, with the exception of dictionaries. To store instances of Dictionary or IdentityDictionary, you need to use a different pattern.

Mapping Dictionaires

When mapping an instance variable that holds a Dictionary, you need to follow a slightly more involved pattern than that for a simple collection. In this situation, three database tables are required, the third being used to hold the links in the dictionary.

For example, let's say our application includes an AddressBook that holds a dictionary of Person objects, each keyed by an alias. That is, the address book can be queried with the string 'Jim', the result being an instance of Person for James Kirk. Of course, a real address book might include several different people named 'Jim', but we'll ignore that for the purposes of illustration.

The class definition for AddressBook is:

```
Smalltalk defineClass: #AddressBook
 superclass: #{Core.Object}
 indexedType: #none
 private: false
 instanceVariableNames: 'id title entries '
 classInstanceVariableNames: ''
 imports: ''
 category: 'My Application'
```

The instance variable entries holds the Dictionary of Person objects. The variable title is intended to hold a string.

As a minimal definition, the Person class looks like this:

```
Smalltalk defineClass: #Person
 superclass: #{Core.Object}
 indexedType: #none
 private: false
 instanceVariableNames: 'id firstName lastName '
 classInstanceVariableNames: ''
 imports: ''
 category: 'My Application'
```

Building the Descriptor System

To model these classes with a `Dictionary` held in `entries`, three tables are needed in the database: `PEOPLE`, `ADDRESS_BOOK`, and `ADDRESS_BOOK_LINKS`. The first two tables represent the domain model classes, while the third is a link table used to model the dictionary of `Person` objects. The `ADDRESS_BOOK` table contains one row for each address book, while the `ADDRESS_BOOK_LINKS` table holds a row for each entry in the address book.

To model these classes, the descriptor system class needs the following basic methods:

```
allTableNames
 ^#('PEOPLE' 'ADDRESS_BOOK' 'ADDRESS_BOOK_LINKS')

 constructAllClasses
 ^(super constructAllClasses)
  add: Person;
  add: AddressBook;
  yourself

 tableForPEOPLE: aTable
 (aTable createFieldNamed: 'first_name' type: (platform varChar: 50)).
 (aTable createFieldNamed: 'last_name' type: (platform varChar: 50)).
 (aTable createFieldNamed: 'id' type: (platform sequence)) bePrimaryKey.

 tableForADDRESS_BOOK: aTable
 (aTable createFieldNamed: 'title' type: (platform varChar: 50)).
 (aTable createFieldNamed: 'id' type: (platform sequence)) bePrimaryKey.

 tableForADDRESS_BOOK_LINKS: aTable
 | personId bookId |
 personId := aTable createFieldNamed: 'person_id' type: platform int4.
 aTable
  addForeignKeyFrom: personId
  to: ((self tableNamed: 'PEOPLE') fieldNamed: 'id').
 bookId := aTable createFieldNamed: 'address_book_id' type: (platform varChar: 50).
 aTable
  addForeignKeyFrom: bookId
  to: ((self tableNamed: 'ADDRESS_BOOK') fieldNamed: 'id').
 aTable createFieldNamed: 'person_key' type: (platform varChar: 30).
```

The tables for `Person` and `AddressBook` classes include primary keys, which are referenced by foreign keys in `ADDRESS_BOOK_LINKS`. Each entry in the address book is thus represented by one row in `ADDRESS_BOOK_LINKS`.

The descriptor system also needs two descriptor methods:

```
descriptorForPerson: aDescriptor
 | personTable |
 personTable := self tableNamed: 'PEOPLE'.
 aDescriptor table: personTable.
 (aDescriptor newMapping: DirectMapping)
  from: #id to: (personTable fieldNamed: 'id').
 (aDescriptor newMapping: DirectMapping)
  from: #firstName to: (personTable fieldNamed: 'first_name').
 (aDescriptor newMapping: DirectMapping)
  from: #lastName to: (personTable fieldNamed: 'last_name').

 descriptorForAddressBook: aDescriptor
 | table linkTable |
 table := self tableNamed: 'ADDRESS_BOOK'.
 linkTable := self tableNamed: 'ADDRESS_BOOK_LINKS'.
 aDescriptor table: table.
 aDescriptor addMapping: (DirectMapping from: #id to: (table fieldNamed: 'id')).
 aDescriptor addMapping: (DirectMapping from: #title to: (table fieldNamed: 'title')).
 aDescriptor addMapping: ((DictionaryMapping new)
  attributeName: #entries;
  referenceClass: Person;
  keyField: (linkTable fieldNamed: 'person_key');
  relevantLinkTableFields: (Array with: (linkTable fieldNamed: 'person_id')))
```

Note that `descriptorForAddressBook:` includes a dictionary mapping, between a key and a `Person` object, named using the two fields in the `ADDRESS_BOOK_LINKS` table.

To complete the descriptor system, we must also add two methods to set up the class models:

```
classModelForPerson: aClassModel
 aClassModel newAttributeNamed: #lastName.
 aClassModel newAttributeNamed: #firstName.
 aClassModel newAttributeNamed: #id.

 classModelForAddressBook: aClassModel
 aClassModel newAttributeNamed: #id.
 aClassModel newAttributeNamed: #title.
 aClassModel newAttributeNamed: #entries collection: Dictionary of: Person.
```

Here, we indicate that the `entries` instance variable is a `Dictionary`.

To create an `AddressBook` and save it in the database:

```
session inUnitOfWorkDo:
 [addresses := AddressBook title: 'contacts'.
 addresses
  at: 'moe' put: (Person first: 'Moe' last: 'Howard');
  at: 'curly' put: (Person first: 'Curly' last: 'Howard');
  at: 'larry' put: (Person first: 'Larry' last: 'Fine').
 session register: addresses].
```

Using this pattern, you can persist `Dictionary` as well as `IdentityDictionary` objects.

In the example above, the key field is already contained within the value mapping: it is the field of the link table of that mapping. We can also map dictionaries when both the key and the value are mappings.

For example, suppose we have a schema with a table for data on whisky bottles, another for pictures of these bottles, and a third for connoisseur sets: collections of whiskies that go well together. We wish to give the `ConnoisseurSet` class an instance variable `bottleViews`, holding a `Dictionary` that maps the pictures to their bottle data. We can define it in the class model, as follows:

```
classModelForConnoisseurSet: aClassModel
 aClassModel newAttributeNamed: #name type: String.
 ...
 aClassModel
  newAttributeNamed: #bottleViews
  dictionaryFrom: Picture
  to: Bottle.
```

... and then in the descriptor:

```
descriptorForConnoisseurSet: aDescriptor
 | pictTable dictMapping keyMapping prodTable |
 ...
```

```
pictTable := self tableNamed: 'PICTURE'.
prodTable := self tableNamed: 'PRODUCT'.
aDescriptor directMapping
 from: #name
 to: (self table fieldNamed: 'NAME').
...

dictMapping := (aDescriptor newMapping: DictionaryMapping)
 attributeName: #bottleViews;
 referenceClass: Bottle.          "dictionary's values are of this class"
keyMapping := (dictMapping newMapping: OneToOneMapping)
  attributeName: #key;
  referenceClass: Picture;     "dictionary's keys are of this class"
  join: (Join
     from: (pictTable fieldNamed: 'BOTTLE')
     to: (prodTable fieldNamed: 'ID')).
dictMapping keyMapping: keyMapping.
```

The join is from the key to the value. It is needed because neither the `ConnoisseurSet` nor the link table between it and the bottle data table (the dictionary’s values) knows it. Therefore the key must be defined by its own mapping.

Index