Glorp Tutorial

by
Roger Whitney
San Diego State University
whitney@cs.sdsu.edu

October 19, 2005

Contents

Introduction

Glorp (Generalized Lightweight Object-Relational Persistence) is an object-to-relational-database mapping layer for Smalltalk. It is available for a number of dialects of Smalltalk including VisualWorks and Squeak. Glorp can be used will a number of relational databases including Oracle, PostgreSQL, SQLServer, Access and SAP.

The examples in this tutorial were run using VisualWorks 7.2, Glorp 0.3.44 through 0.3.111 and PostgreSQL 7.4.

Relational databases are often used for persistence of data for programs. Smalltalk code consists of interacting objects. Relational databases consist of tables of data. The tables are often normalized. The goal of GLORP is to make the use of relational databases with Smalltalk as transparent as possible.

Many of the concepts used by GLORP are discussed in Martin Fowler's book Patterns of Enterprise Application Architecture. For those that wish to learn more about object-to-relational mappings references to relevant sections of the book are given in this tutorial.

A Database

Before you can use GLORP you need to have a relational database to work with. This tutorial does not cover installing and running a relational database. You also need to have the drivers for that database loaded in the Smalltalk image.

Loading GLORP

To get the latest version of GLORP use the CINCOM public repository. Load the bundles Glorp and GlorpTest. The later contains many unit tests that are useful in learning about Glorp features. The examples in this tutorial are slight variations on examples found in the unit tests.

If you don't need the latest version of GLORP you can load the version that comes with VisualWorks. You will find it in the preview directory of the VW installation. First load the parcel GlorpVWPort.pcl then load the parcels Glorp.pcl and GlorpTest.pcl.

Running GLORP Unit Tests

Before you can run the GLORP unit tests you need to configure them to use your database. To do this you need to edit the correct class method in the class GlorpDatabaseLoginResource, which is in the GlorpDBTests package in the GlorpTest bundle. The accessing protocol of the class methods in GlorpDatabaseLoginResource contains methods for default logins for different databases. Find the correct one for your database edit it with the correct database information. For this tutorial I used a PostgreSQL database on my local machine, so I edited defaultPostgreSQLLocalLogin as below. Once the method contains the correct username, password and connection data execute the comment in the method assigning DefaultLogin.

Now you are ready to run the GLORP unit tests. The tests are regular SUnit tests, so run them with your favorite SUnit tool. There are over 300 tests so it will take a while to run them all.

```
GlorpDatabaseLoginResource class>>defaultPostgreSQLLocalLogin
    "To set the default database login to PostgreSQL, execute the following statement."
    "DefaultLogin := self defaultPostgreSQLLocalLogin."

    ^(Login new)
        database: PostgreSQLPlatform new;
        username: 'whitney';
        password: 'topSecret';
        connectString: 'localhost:5432_test'.
```

Simplest Possible GLORP Example

It is possible in GLORP to send an SQL statement to the database. If this all you wish to do there is no point in using GLORP, just use the database drivers for your database. The following example is useful to determine if GLORP can talk to the database. If you are not using PostgreSQL you need to change the connectString and the SQL set to the database. The connectString is the connect string used by the database driver for your database. For PostgreSQl the connection string has the form machineNameOrIP:portNumber_database. The SQL string needs to be something that you know will work on your database.

```
login := Login new database: PostgreSQLPlatform new;
    username: 'usernameHere';
    password: 'passwordHere';
    connectString: '127.0.0.1_test'.
```

Once we have the needed information we can log on to the database.

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

Once we are logged on we can use an accessor to execute SQL statements.

```
result := accessor basicExecuteSQLString: 'select 1 + 1'.
result next first

accessor logout.
```

The last statement ends the connection to the database.

Simple Object One Table Example

For the first example we will store Person objects in a single table. The Person class is given below. It has two instance variables and two methods to initialize the instance variables.

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

 Person class methods

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


 Person instance methods

setFirst: firstNameString last: lastNameString
    firstName := firstNameString.
    lastName := lastNameString

Person>>firstName: anObject
    firstName := anObject
```

Now we need to define the table and the mapping between the table and objects of type Person. This is done in a descriptor class. The class must be a subclass of Glorp.DescriptorSystem. Here is the class declaration. One of the methods will use the Glorp.DirectMapping class so it is imported.

```
Smalltalk defineClass: # GlorpTutorialDescriptor
    superclass: #{Glorp.DescriptorSystem}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            Glorp.DirectMapping
            '
    category: 'GlorpExperiments'
```

To define the mapping between the table and objects we need five methods: allTableNames, constructAllClasses, tableForXXX:, descriptorForYYY:, classModelForYYY.

The method allTableNames just returns an array of all the tables that the descriptor defines. In this example there is just one table. The method is given below. Keep in mind that while the database table names may not be case sensitive, Smalltalk strings and method names are case sensitive. So all references in the descriptor to the table must use the same case for the table name.

```
GlorpTutorialDescriptor >>allTableNames
    ^#( 'PEOPLE' )
```

Now we need to define the table. This is done in a method called tableForXXX: where XXX is replaced with the table name. So in our example the name of the method is called tableForPEOPLE:. The method is given below. We don't use SQL directly to define the table. This will allow GLORP to generate the correct SQL for the what ever database we use. The means that we need to learn the classes and methods to define tables. This will be done a bit later. Here is the method for the PEOPLE table.

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

Now we need to tell the descriptor about the class whose objects will be stored in the table. First we have the method constructAllClasses that just lists the classes involved in tables defined in the descriptor. At this point we only have one class. Here is the method:

```
GlorpTutorialDescriptor >>constructAllClasses
    ^(super constructAllClasses)
        add: Person;
        yourself
```

Now we need to define the mapping between the table and the class. This is done in a method called descriptorForYYY: where YYY is replaced with the name of the class. In our case the method name is descriptorForPerson:. For each instance variable we create a DirectMapping between the instance variable and a column.

```
GlorpTutorialDescriptor >>descriptorForPerson: aDescriptor

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

Each class needs a model. This is given in the method called classModelForYYY: where YYY is the name of the class. A class model provides information about the class. In an example this simple one does not need to provide the class model. GLORP will determine the correct values. We will see examples later where the class model is useful.

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

Now we are ready to connect to the database, create the table and add data to the table.

First we create a Login object with the correct database type and login information. Since I am connecting to a PostgreSQL database I use the PostgreSQLPlatform. See the subclasses of DatabasePlatform for the databases supported by GLORP. This example will use a PostgreSQL database called glorpTests. As GLORP will not create databases this database was created using standard PostgreSQl tools.

```
login := Login new database: PostgreSQLPlatform new;
    username: 'whitney';
    password: 'foo';
    connectString: '127.0.0.1_glorpTests'.

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

Once we have an accessor the database we can interact with the database. If we just want to execute SQL strings then we can do it directly on the accessor. However, the power of GLORP is in the high level interaction with the database. All the high level interaction with the database is done through a session. So here is how to get a session.

```
session := GlorpSession new.
session system: (GlorpTutorialDescriptor   forPlatform: login database).
session accessor: accessor.
```

Now we are ready to create the tables defined in the GlorpTutorial. One does not have to create the tables using Glorp. The tables can be created using other tools and then used by Glorp. Indeed there are some database constraints that currently cannot be created using Glorp. If you need such constraints then you need to create the database tables via other means. Regardless of how you create the database tables the Glorp descriptor needs to contain the table definitions.

```
session inTransactionDo:
    [session system allTables do:
        [:each |
        accessor
            createTable: each
            ifError: [:error |Transcript show: error messageText]]].
```

This code will generate the correct SQL to create the tables and execute it on the database. By default GLORP echoes it interaction with database in the Transcript. So you should see the SQL statement generated. Now we will add some data to the table. The following will add the data from a Person object to the table. A UnitOfWork[1] stores the objects and commits them to the database when the UnitOfWork is committed. Not shown here, but one can start transactions and commit them or roll them back.

```
session beginUnitOfWork.
person := Person first: 'Pete' last: 'Frost'.
session register: person.
session commitUnitOfWork.
```

Now to read some data from the database. We do not have much to work with so the following code will read all people from the table.

```
people := session readManyOf: Person .
```

Now to add a few more people to the table:

```
session beginUnitOfWork.
#( 'Sam' 'Jose' 'Rose' )
    with: #( 'Olson' 'Chan' 'Menon' )
    do: [:first :last |
        person := Person first: first last: last.
        session register: person].
session commitUnitOfWork.
```

The SQL generated when this was committed is below.

[1] See Unit of Work pattern pp 184-189 in Fowler [2003]

```
Begin Transaction
INSERT INTO PEOPLE (first_name,last_name)  VALUES ('Rose','Menon');
INSERT INTO PEOPLE (first_name,last_name)  VALUES ('Jose','Chan');
INSERT INTO PEOPLE (first_name,last_name)  VALUES ('Sam','Olson')
(0.032 s)
Commit Transaction
```

Now we can select data from the table based on some selection criteria.

```
foundPerson := session readOneOf: Person where: [:each | each firstName = 'Jose']
```

The block appears to be sending a message to a person object, but is not. The block is used to generate the SQL statement given below. In the block “each firstName” refers to the name of an instance variable of the Person class. Syntax and the semantics of the where: block are covered later.

```
FROM PEOPLE t1
   WHERE (t1.first_name = 'Jose')
```

A UnitOfWork will keep track of the changes we make in objects. The following will retrieve a person, change the first name of person and then write the changes back to the database.

```
session beginUnitOfWork.
foundPerson := session readOneOf: Person where: [:each | each firstName = 'Jose'].
foundPerson firstName: 'RamJet'.
session commitUnitOfWork
```

When we start a UnitOfWork it records all the objects we read from the database. When the UnitOfWork is committed it writes all the changed objects back to the database.

Once we commit a UnitOfWork it clears its list of objects that it is tracking. If we want the UnitOfWork to retain its list of objects use commitUnitOfWorkAndContinue or saveAndContinue instead of commitUnitOfWork.

We can also rollback a UnitOfWork.

```
session beginUnitOfWork.
foundPerson := session readOneOf: Person where: [:each | each lastName = 'Chan'].
foundPerson firstName: 'Ham'.
session rollbackUnitOfWork.
```

When you rollback a UnitOfWork all the objects in the UnitOfWork are also rolledback. That is all the changes made to the objects are undone. In the above example the object foundPerson has it original first name.

Since it is fairly common to always commit a unit of work there is a short hand version.

```
session inUnitOfWorkDo:
    [foundPerson := session readOneOf: Person where: [:each | each firstName = 'Jose'].
    foundPerson firstName: 'RamJet']
```

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.

Some Table Details

In the tableForPEOPLE: method we define the table to hold the data from Person objects.

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

The above example leads to two questions:

- What types can we declare a column and
- What modifiers (NULL, NOTNULL, etc) can we use on a column

Defining Column Types

In the method tableForPEOPLE: aTable is an instance of Glorp.DatabaseTable. Its method createFieldNamed:type: adds a Glorp.DatabaseField to the table. This represents a column of the table. The instance variable “platform” is an instance of a subclass of Glorp.DatabasePlatform. In our example it is an instance of PostgreSQLPlatform. DatabasePlatform and its subclasses define the types we can declare a column. DatabasePlatform contains the following methods to define types of a column.

blob	char	char:
character	clob	datetime
inMemorySequence	int4	integer
sequence	smallint	text
timestamp	varChar	varchar
varChar:	varchar:	

PostgreSQLPlatform inherits the methods above and contains the following methods to define types of a column. At least with the PostgreSQLPlatform the varchar method with no arguments did not produce useable SQL.

bigint	blob	boolean	clob	date	decimal
double	float	float4	float8	int	int2
int4	int8	integer	numeric	real	sequence
serial	smallint	time	timestamp	timetz	varchar

Most of these are fairly standard database types and map to obvious Smalltalk types.

Column or Type Modifiers

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

When defining a table in Glorp we can add modifiers to columns. In the example above the last_name column is made the primary key. The Glorp.DatabaseField class defines the following methods that modify columns:

beLockKey	beNullable:	bePrimaryKey
defaultValue:	isUnique:	type:

beNullable: A false value means the column cannot be null. The default value is true, that is the column can be null.

isUnique: A true value means all values in the column must be unique. The default value is false.

defaultValue: sets the default value of the column if the instance variable mapped to the column is nil. At least when using the PostgreSQLPlatform default values don't use the SQL default value. Glorp inserts the default value before sending the data to the database.

beLockedKey I have no idea what this does.

bePrimaryKey Declares a column or columns to be the primary key of the table. More on this in the next section.

type: Sets the type of the column. In the examples in this tutorial this method is not used as the types are set with createFieldNamed:type: method on a table.

Here is an example of using the different modifiers.

```
tableForCOLUMN_MODIFIERS: aTable
    (aTable createFieldNamed: 'a' type: platform sequence) bePrimaryKey .
    (aTable createFieldNamed: 'b' type: (platform varchar: 10)) defaultValue: 'cat'.
    (aTable createFieldNamed: 'c' type: platform boolean) beLockKey.
    (aTable createFieldNamed: 'd' type: platform float4) beNullable: false.
    (aTable createFieldNamed: 'e' type: platform real) isUnique: true.
    (aTable createFieldNamed: 'f' type: platform float) isUnique: false.
    (aTable createFieldNamed: 'g' type: (platform int))
        defaultValue: 5;
        beNullable: false
```

Here is the SQL generated by GLORP from the above table definition.

```
CREATE TABLE COLUMN_MODIFIERS (
    A serial NOT NULL ,
    B varchar(10) NULL ,
    C boolean NULL ,
    D float4 NOT NULL ,
    E float4 NULL UNIQUE,
    F float4 NULL ,
    G int4 NOT NULL ,
    CONSTRAINT COLUMN_MODIFIERS_PK PRIMARY KEY  (a),
    CONSTRAINT COLUMN_MODIFIERS_UNIQ UNIQUE  (a)
)
```

Primary Keys

In general Glorp will not insert an object into a table that does not have a primary key. Link tables and embedded values are exceptions.

Explicit Declaration of Primary Key

In the example we used a short cut to declare a column to be a primary key.

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

One can also explicitly indicate that a column is a primary key on a table as shown below.

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

MultiColumn Primary Key

If the primary key spans multiple columns in the table, just make each column a primary key as done below.

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

Auto-Generated Primary Key

It is common to generate a number to be the primary key for a table. We will use a sequence in the database to do this for the Person example. We add an id instance variable to the Person class so it now has three instance variables.

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

We modify the descriptorForPerson: and tableForPEOPLE: methods in the GlorpTutorialDescriptor to include the id information. In the table definition we make the ID column a sequence and the primary key.

```
GlorpTutorialDescriptor >>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').

GlorpTutorialDescriptor >>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)).
```

Now we can create the table with the code below. Note the addition of the create sequences. With PostgreSQL sequences are not created explicitly so the code for the sequences is not needed, but is shown for other databases.

```
login := Login new database: PostgreSQLPlatform new;
    username: 'whitney';
    password: 'earl';
    connectString: '127.0.0.1_glorpTests'.

accessor := DatabaseAccessor forLogin: login.
accessor login.

session := GlorpSession new.
session system: (GlorpTutorialDescriptor   forPlatform: login database).
session accessor: accessor.
session inTransactionDo:
        [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]]]
```

The above code does create the table. It generates the following SQL:

```
Begin Transaction
CREATE TABLE PEOPLE (ID serial NOT NULL ,FIRST_NAME varchar(50) NULL
,LAST_NAME varchar(50) NULL , CONSTRAINT PEOPLE_PK PRIMARY KEY  (ID),
CONSTRAINT PEOPLE_UNIQ UNIQUE  (ID))
Commit Transaction
```

Once the table is created we can add a Person object using:

```
session beginUnitOfWork.
person := Person first: 'Roger' last: 'Whitney'.
session register: person.
session commitUnitOfWork
```

When a session has a registered Person object with a nil value for the id instance variable, GLORP retrieves the next value in the id sequence and sets the id. Once this is done the object is added to the table. If you create a new Person object with an id, GLORP will try to add it to the table when it is registered with a session and committed. However if you read the object from the database, modify it and then commit it in a unit of work the correct row of the database is updated.

Direct Instance Variable Access or Accessor Methods

When writing data to and reading data from a database GLORP has to access data in an object's instance variables. GLORP's default is to access instance variables directly. That is not use the classes accessor methods to access the instance variables. There are times when GLORP should use the accessor methods to read or set instance variables. To show how this is done we first need the accessor methods in the Person class.

```
Person>>firstName: anObject
    firstName := anObject

Person>>firstName
    ^firstName

Person>>lastName: anObject
    lastName := anObject

Person>>lastName
    ^lastName

Person>>id: anObject
    id := anObject

Person>>id
    ^id
```

We can specify which instance variables we want to be accessed by setter/getter methods in the class model for the class. Below we indicate that GLORP is to access the instance variable firstName via the methods Person>>firstName and Person>>firstName:.

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

To tell GLORP to use accessor methods for all instance variables in all classes registered in the GlorpTutorialDescriptor use the method DescriptorSystem>>useDirectAccessForMapping:

```
session := GlorpSession new.
descriptor := GlorpTutorialDescriptor   forPlatform: login database.
Descriptor useDirectAccessForMapping: false.
session system: descriptor.
session accessor: accessor.
```

The above method allows one to determine at runtime which method of access to use on all classes. If one wants to insure that accessor methods are always used you can override the method useDirectAccessForMapping.

```
GlorpTutorialDescriptor>>useDirectAccessForMapping
    ^false
```

Reading, Writing, Deleting Data

Reading, writing, updating and deleting data are common operations done with a database. Methods to perform these operations are found in api/query protocol of the GlorpSession class.

Reading

A session has the following methods for reading data or objects from the database

readOneOf:	readOneOf:where:
readManyOf:	readManyOf:where:
readManyOf:limit:	readManyOf:where:limit
readManyOf:orderBy:	

The first argument of each method is a class of the objects you wish to read. The where: argument is a block with one argument. This block is used to generate the SQL select statement, but allows you to use objects and methods to specify the selection criteria. The readOneOf: methods will return the first record that satisfies the selection criteria. The readManyOf methods return an array of all objects that satisfy the selection criteria. The limit: argument can be used to restrict readManyOf: to return the first N objects that satisfy the selection criteria. There does not seem to be a simple way to read the objects starting with the N+1 object, so you need to use Query objects to do that (see the section **Advanced Reading**).

```
session readManyOf: Person limit: 2.
session readOneOf: Person where: [:each | each  firstName  = 'Sam'].
session readManyOf: Person orderBy: [:each | each lastName].
```

Selection Criteria in where clause

The where: block of the read statement is used to generate SQL that is executed on the database. That is these blocks are a language to specify select statements. As a result there are some restrictions on what can be done in these blocks. The following description is based on examining senders of the messages readManyOf:where: in Glorp's test cases.

The first restriction is on the methods that can be sent to the argument of the block. Messages sent to the block argument must be:

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

Since the block is used to generate SQL, there is no need for accessor methods for the instance variables. So for the following statement to work there is no need for a firstName method in the Person class.

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

The messages = or <> can be used to read objects based on objects previously read from the database. In the following example we read the first 4 people who are not Sam.

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

The SQL generated by the last statement is:

```
SELECT t1.first_name, t1.last_name, t1.id
 FROM PEOPLE t1
 WHERE (t1.id <> 2)
```

That is [:each | each <> sam] generates a comparison based on the primary key of the object sam. If the sam object's primary key (id) is nil then select statement will be:

```
SELECT t1.first_name, t1.last_name, t1.id
 FROM PEOPLE t1
 WHERE (t1.id IS NOT NULL)
```

The notNIL and isNIL messages are not particularly useful at this level. The notNIL or isNIL generates IS NULL or IS NOT NULL tests on the primary key of the table. So one either gets the entire table or no elements. The following statement will read all the person objects in the database.

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

The second restriction is what messages can be sent to instance variables in the block. Given that we currently only have one table the messages we can send are:

- Binary comparison operators: <, >, =, ~=, >=, <=, <>
- like:
- isNIL, notNIL
- Methods in the api protocol of Glorp.ObjectExpression

We will add to this list when we have objects that are more complex and span multiple tables in the database. See the end of the section **Association (Tie)Tables – Many-to-many**.

We have already seen examples of using the binary operators. Here are a few more:

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

The like method generates SQL select statements using SQL's LIKE. The following reads all person objects whose first name starts with S. The character "_" matches any single character. The character "%" matches any sequence of characters. In some databases LIKE is case sensitive (PostgreSQL for example) in others it is not (MySQL for example).

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

This statement generates the following SQL:

```
SELECT t1.first_name, t1.last_name, t1.id
 FROM PEOPLE t1
 WHERE (t1.first_name LIKE 'S%')
```

The following statement generates the NOT LIKE clause.

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

The methods isNIL and notNIL are used to find object where a given column in the database is null or not. The following statement returns all people in the database whose first name is not null in the

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

These operations can be combined with or, and negation messages. Here are a few such statements.

```
session
    readManyOf: Person
    where: [:each | (each  firstName notNIL) & ( each lastName like: 'Whit%')].
session
    readManyOf: Person
    where: [:each | (each  firstName notNIL) and: [each lastName like: 'Whit%']].
```

The methods in the api protocol of Glorp.ObjectExpression are rather specialize so will not be covered in this tutorial. Look for senders of those messages in the Glorp test cases to see how they are used.

Deleting

You can delete an object via the GlorpSession>>delete: and a collection of objects via GlorpSession>>deleteAll:. Below is an example of deleting an object. First we get the object from the database. The delete: method removes the object's data from the database.

```
foundPerson := session readOneOf: Person where: [:each | each  firstName  = 'Roger'].
session delete: foundPerson
```

When done this way the object is deleted immediately. If one does a delete inside a UnitOfWork then you can rollback a delete.

```
session beginUnitOfWork.
foundPerson := session readOneOf: Person where: [:each | each  firstName  = 'Roger'].
session delete: foundPerson.
session rollbackUnitOfWork.       "or session commitUnitOfWork"
```

The delete method on a session checks to see if it is in a UnitOfWork. If it is the object to delete is registered with the UnitOfWork. If it is not in a UnitOfWork a UnitOfWork is started, the object registered and then the UnitOfWork is committed.

Rereading

When we read data from a database the data exists in two places: in the database and in memory. If another thread or program modifies the database the data in memory will be different than what is in the database. The method GlorpSession>>refresh: will update an object with the current data in the database.

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

"some time passes"

session refresh: foundPerson
```

Read Only Mappings

In our descriptor for a class we define mappings from the instance variables of a class to columns in a table. The default is to have the mapping both read and write. That is the mapping can be used to read data from and write data to the column in the table. There are times when one wants the mapping to be read only. That is the mapping can only be used to read data from the column, but cannot write to the column. In the descriptor below the mapping for the first name is made read only.

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

While this demonstrates how to make a mapping read only, in this example one would want either all mappings read only or all mappings read-write. As it stands now the first name field would null for the newly added row when one tries to add a person object to the database with:

```
session beginUnitOfWork.
person := Person first: 'Roger' last: 'Whitney'.
session register: person.
session commitUnitOfWork
```

If one makes all the mappings read only then when you try to add a person no rows will be added to the table. Read only mappings are more useful with more complex mappings.

Transactions

GLORP supports transactions. Operations done inside of a transaction can be committed or rolled back as is normal with transactions. There are two ways to explicitly use a transaction. The inTransactionDo: method will wrap the block in a transaction. If an exception is raised in the bock the transaction is rolled back. Otherwise the transaction is committed. The block can have either no or one argument. If the block does have an argument, it is passed a reference to the session.

```
session inTransactionDo: [ put code here]

session inTransactionDo: [:aSession | put code here]
```

There are times when one needs more control over committing or rolling back a transaction. In that case one explicitly starts the transaction with the beginTransaction and either commits with commitTransaction or rolls back with rollbackTransaction.

```
session beginTransaction.
Code
x < 10
    ifTrue:[ session commitTransaction]
    ifFalse: [ session rollbackTransaction]
```

A unit of work automatically wraps it self in a transaction, so there is no need to explicitly run a unit of work in a transaction. This explains why none of the examples so far has explicitly used a transaction.

Some Common Problems

There are two common problems that can cause you a lot of frustration and wasted time. In a production system these will not be an issue. However when you are learning GLORP you will be making lots of little changes to see how things work. If you find that your changes are not reflected in the behavior of the system come back and read this section again.

Cached Values

The first problem is caused by cached values. When you create a new session, as done below, the descriptor is read and its data is cached. If you then make changes to the descriptor class, those changes will not be reflected in any open session. You need to start a new session for the changes in the descriptor class to take effect. So if you find yourself making changes to the descriptor class but those changes are not reflected in your code, use a new session.

```
session := GlorpSession new.
session system: (GlorpTutorialDescriptor   forPlatform: login database).
session accessor: accessor.
```

Creating Tables

Once you have created a table you need to delete the table before creating it a second time. When learning GLORP you are likely to change table structure a lot. If you tell GLORP to create the tables it will appear to do so. The proper SQL is sent to the database and no errors are raised. However the tables are not recreated. There is a very subtle way in which GLORP informs you that the tables were not created. If the tables were created the GLORP log in the Transcript will include the time it took to create the tables. Here is what you will see in the Transcript if the tables are actually created:

```
Begin Transaction
CREATE TABLE PEOPLE (ID serial NOT NULL ,FIRST_NAME varchar(50) NULL
,LAST_NAME varchar(50) NULL , CONSTRAINT PEOPLE_PK PRIMARY KEY  (id),
CONSTRAINT PEOPLE_UNIQ UNIQUE  (id))
(1.116 s)
Commit Transaction
```

Note the (1.116s) after the SQL. Now below is the GLORP log when the tables already exist so the CREATE TABLE has no effect. There is no creation time listed.

```
Begin Transaction
CREATE TABLE PEOPLE (ID serial NOT NULL ,FIRST_NAME varchar(50) NULL
,LAST_NAME varchar(50) NULL , CONSTRAINT PEOPLE_PK PRIMARY KEY  (id),
CONSTRAINT PEOPLE_UNIQ UNIQUE  (id))
Commit Transaction
```

While learning GLORP changing table structure will be common. Here is Smalltalk code which will first drop the tables then create them. Do not put the 'accessor dropTables: …' code in the same transaction that creates the tables. If the tables do not exist the dropTables: method will abort the transaction. The dropTables: method wraps itself in a Transaction so there is no need to explicitly put it in a Transaction.

```
session := GlorpSession new.
session system: (GlorpTutorialDescriptor forPlatform: login database).
session accessor: accessor.
accessor dropTables: session system allTables.
session inTransactionDo:
        [
        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]]]
```

Complex Objects & Tables
Instance Variable with a table - One-to-one

We will look at an example where an object has data in two different tables in the database. We add an address to the Person class. Each person will only have one address. An Address is a class with instance variables for a street and a city. Here are the declarations of each class:

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

```
Smalltalk defineClass: #Address
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'street city id '
    classInstanceVariableNames: ''
    imports: ''
    category: 'GlorpExperiments'
```

In the database we will use two tables, PEOPLE and ADDRESSES to store the data. Two tables are used since more than one person may have the same address. First we need to create the GLORP descriptor for the database. Here is the definition of the descriptor class:

```
Smalltalk defineClass: #GlorpTutorialDescriptor
    superclass: #{Glorp.DescriptorSystem}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            Glorp.*
            '
    category: 'GlorpExperiments'
```

Now we need to provide the list of classes and tables. Here are the relevant methods:

```
GlorpTutorialDescriptor >>allTableNames
    ^#( 'PEOPLE' 'ADDRESSES' )

GlorpTutorialDescriptor >>constructAllClasses
    ^(super constructAllClasses)
        add: Person;
        add: Address;
        yourself
```

The definition of the ADDRESSES table is straightforward.

```
GlorpTutorialDescriptor >>tableForADDRESSES: aTable
    aTable createFieldNamed: 'street' type: (platform varChar: 50).
    (aTable createFieldNamed: 'city' type: (platform varChar: 50)).
    (aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey
```

In the PEOPLE table we need to add a foreign key[2]. Here is how it is done.

```
GlorpTutorialDescriptor >>tableForPEOPLE: aTable
    | addressId |
    (aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey.
    (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').
```

Each class also needs a descriptor entry. The Address class has the same structure as the Person class in the last example. So the descriptor does not have anything new.

```
GlorpTutorialDescriptor >>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: #id
        to: (table fieldNamed: 'id').
```

In the Person class we need information about the address. The last two entries do this. The OneToOneMapping tells GLORP to use a different table (via a one-to-one map) to get the data for address. Note we do not provide information about which table to get the data for the address instance variable. That information will be added in a different location.

[2] See Foreign Key Mapping pattern pp 236-247 in Fowler [2003]

```
GlorpTutorialDescriptor >>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: OneToOneMapping)
            attributeName: #address.
```

We still need to let GLORP know how to deal with the address instance variable in the Person class. This is done using a class model. In the method we provide information about each instance variable.

```
GlorpTutorialDescriptor >>classModelForAddress: aClassModel
    aClassModel newAttributeNamed: #id.
    aClassModel newAttributeNamed: #street.
    aClassModel newAttributeNamed: #city.

GlorpTutorialDescriptor >>classModelForPerson: aClassModel
    aClassModel newAttributeNamed: #id.
    aClassModel newAttributeNamed: #firstName.
    aClassModel newAttributeNamed: #lastName.
    aClassModel newAttributeNamed: #address type: Address.
```

Of interest is “address”. The line:

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

tells GLORP that the instance variable address holds an object of type Address. Since the descriptor contains information about the Address class, this is how GLORP knows how to map the address instance variable of Person to the database.

Now we can log on to the database and create the tables.

```
login := Login new database: PostgreSQLPlatform new;
    username: 'foo';
    password: 'bar';
    connectString: '127.0.0.1_glorpTests'.
```

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

session := GlorpSession new.
session system: (GlorpTutorialDescriptor forPlatform: login database).
session accessor: accessor.

accessor dropTables: session system allTables.
session inTransactionDo:
        [
        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]]]
```

Once the tables are created we can add some data. Here we create a Person object with an address and add it to the database.

```
session beginUnitOfWork.
person := Person first: 'Sam' last: 'Whitney'.
address := Address new.
address
    street: 'Maple';
    city: 'SD'.
person address: address.
session register: person.
session commitUnitOfWork
```

Just for the record here is the SQL generated during the above transaction.

```
Begin Transaction
select nextval('ADDRESSES_id_SEQ') from ADDRESSES limit 1
select nextval('PEOPLE_id_SEQ') from PEOPLE limit 1
INSERT INTO ADDRESSES (street,city,id)  VALUES ('Maple','SD',2)
(0.004 s)
INSERT INTO PEOPLE (id,first_name,last_name,address_id)  VALUES (2,'Sam','Whitney',2)
(0.003 s)
Commit Transaction
```

With some data in the database we can make queries on the data. Here is a simple query. Since the statement just reads data there is no need to put this statement in a transaction or unit of work.

```
foundPerson := session readOneOf: Person where: [:each | each address city  = 'SD'].
```

The query uses the following SQL to get the data from the database:

```
SELECT t1.id, t1.first_name, t1.last_name, t1.address_id
 FROM (PEOPLE t1 INNER JOIN ADDRESSES t2 ON (t1.address_id = t2.id))
 WHERE (t2.city = 'SD')
```

Alert readers will note that only the id is read from the ADDRESSES table. The Person object created by the query has a proxy[3] for its instance variable. The data for the address is not fetched from the database until it is actually used. After adding the accessor methods to the People and Address classes the following statement:

```
foundPerson address street
```

will trigger the proxy to fetch the address object. The SQL used to do that is:

```
SELECT t1.street, t1.city, t1.id
 FROM ADDRESSES t1
 WHERE (t1.id = 1)
```

Using a unit of work any change to the person object will be saved to the database. Here we change the address object in the person object. When the unit of work is committed the address data is updated.

```
session beginUnitOfWork.
foundPerson := session readOneOf: Person where: [:each | each address city  = 'SD'].
foundPerson address street: 'Pine'.
session commitUnitOfWork.
```

Here is the SQL used to update the data. Note that GLORP is making an unneeded request for the next value in the ADDRESSES is sequence.

```
Begin Transaction
select nextval('ADDRESSES_id_SEQ') from ADDRESSES limit 1
UPDATE ADDRESSES SET city = 'SD',street = 'Pine' WHERE id = 1
(0.093 s)
Commit Transaction
```

[3] See Lazy Load pattern pp 200-214 in Fowler [2003]

Once we are done with the session we logout:

```
accessor logout
```

Duplicate Rows

Adding a Person object to the database also adds the person's address if the address is new. If two people have the same address it is possible to add the same address twice. For example suppose we first do:

```
session inUnitOfWorkDo:
    [roger := Person first: 'Roger last: 'Whitneyy'.
    address := Address street: 'Campanile Way' city: 'SD'.
    roger address: address.
    session register: roger].
```

And at some later time we do:

```
session inUnitOfWorkDo:
    [sam := Person first: 'Sam' last: 'Hinton'.
    address := Address street: 'Campanile Way' city: 'SD'.
    sam address: address.
    session register: sam].
```

Now the address table will have two rows with the same data. If you do not want this to occur care is needed. You should read the address from the database and then add it to the person object.

Orphan Rows

Although adding a person to the database will add its address to the database if needed, deleting a person does not delete its address. That is if one first does:

```
session inUnitOfWorkDo:
    [roger := Person first: ‘Roger last: ‘Whitneyy’.
    address := Address street: 'Campanile Way' city: 'SD'.
    roger address: address.
    session register: roger].
```

And then does

```
    roger := session readOneOf: Person
                where: [:each | each firstName = 'Roger'].
    session delete: roger.
```

The address associated with roger is still in the address table. So one needs to explicitly delete each object from the database.

PseudoVariables

In our descriptor we can define pseudovariables for a class. These are variables that do not exist in the class, but we can use in a glorp query about the class. Below we define a pseudovariable addressId for the address_id column in the People table.

```
GlorpTutorialDescriptor >>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: DirectMapping)
        fromPseudoVariable: #addressId
        to: (personTable fieldNamed: 'address_id').
    (aDescriptor newMapping: OneToOneMapping)
            attributeName: #address
```

In a query we can then use the psuedovariable as if it were an instance variable. Here is an example use of the psuedovariable.

```
roger := session readOneOf: Person where: [:each | each addressId = 1].
```

In general I prefer not to deal with database id's in Smalltalk code as this example does. It is rather dangerous to assume that particular data is in the k'th row of a table.

Collections as Instance Variable - One-to-many

Often an object will contain a collection. The way this is handled in a database is to create a separate table for the collection. Each row in the table contains a foreign key indicating who owns the row. GLORP handles this with a one-to-many map. To illustrate this we give the Person class a collection of email addresses. An Email Address has a username and host. We create EMAIL_ADDRESSES table with columns:

Column	Description
id	Primary Key for table
user_name	User name of the email address
host	The email host
person_id	Foreign key to person table

Here is the definition of the EmailAddress class.

```
Smalltalk defineClass: #EmailAddress
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'userName host id '
    classInstanceVariableNames: ''
    imports: ''
    category: 'GlorpExperiments'
```

EmailAddress class method

```
name: aNameString host: aHostString
    ^(super new)
        userName: aNameString;
        host: aHostString
```

The class also has standard accessor methods to set and get userName and host.

The Person class definition:

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

Person class method

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

Person instance methods

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

addEmailAddress: anEmailAddress
    emailAddresses add: anEmailAddress

emailAddresss
    ^emailAddresses
```

The Person class also has get and set methods for lastName and firstName.

To save space the example does not include the address added in the one-to-one section. Since the only thing new here is the one-to-one mapping the example is given in full without comment. Here is the Descriptor:

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

```
Smalltalk defineClass: #GlorpTutorialDescriptor
    superclass: #{Glorp.DescriptorSystem}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            Glorp.*
            '
    category: 'GlorpExperiments'

allTableNames
    ^#( 'PEOPLE'  'EMAIL_ADDRESSES')

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

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

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

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').
```

```
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.

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').

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)).
```

Order of the Email Addresses

When the email addresses are read for a person we can determine the order in which they are read from the database. This is done by telling the mapping the order to use when reading values. In the descriptor for the person we indicate that the addresses are to be sorted by the userName. The ordering is done on the database. Multiple orderBy: messages can be sent to a mapping.

```
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: ToManyMapping)
        attributeName: #emailAddresses;
        orderBy: #userName
```

The read order can be specified by indicating the table and column explicitly. There is no reason to do this in the current example. However when we have association (tie or link) tables a mapping will involve three tables. In that case being able to indicate the table and column can be useful. Here is how one specifies the table and column.

```
(aDescriptor newMapping: ToManyMapping)
    attributeName: #emailAddresses;
    orderBy: [:each | (each getTable: 'EMAIL_ADDRESSES') getField: 'user_name']
```

Specifying the Type of Collection

When one reads a person from the database as below, the email addresses in the person object are stored in an ordered collection.

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

As of GLORP 0.3.111 one can specify in the class model which type of collection GLORP will use. Below we indicate that a SortedCollection should be used. One can use all the standard collection classes except a Dictionary, which requires extra information. Using a dictionary is covered in the next section.

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

Note this feature does not work in versions of GLORP before 0.3.111 unless one does a lot more work. See GlorpCollectionTypesDescriptorSystem>> descriptorForGlorpThingWithLotsOfDifferentCollections: in the bundle GlorpTest for examples.

Dictionaries as Instance Variables - One-to-many

Mapping a dictionary to a database table is slightly more complex then mapping an ordered collection. To illustrate this we will create an address book that stores instances of people. The address book will store people in a dictionary. Keys will be a string that users associate a particular person with. Here is the AddressBook class.

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

AddressBook class methods

```
title: aString
    ^super new setTitle: aString
```

AddressBook instance methods

```
at: aString
    ^entries at: aString

at: aString put: aPerson
    entries at: aString put: aPerson

title
    ^title

setTitle: aString
    title := aString.
    entries := Dictionary new.
```

The Person class just has the instance variables 'firstName, lastName and id. The class definition is below. You can add whatever methods you want to the class.

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

We need three tables in the database. First we need one to hold the Person object data. Second we need a table to for address books. Each row in the ADDRESS_BOOK table represents one address book. In this case the table just has two columns, one for a primary key and another for the title of the address book. The third table, ADDRESS_BOOK_LINKS, is an association (link or tie) table. Each row in the association table represents one entry in an address book.

```
Smalltalk defineClass: #GlorpTutorialDescriptor
    superclass: #{Glorp.DescriptorSystem}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            Glorp.*
            '
    category: 'GlorpExperiments'

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

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

Note how we indicate that the entries instance variable is a dictionary.

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

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

The address book descriptor includes a dictionary mapping which ties the data from the other two tables to the address book.

```
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: ((BasicDictionaryMapping new)
                attributeName: #entries;
                referenceClass: Person;
                keyField: (linkTable fieldNamed: 'person_key');
                relevantLinkTableFields: (Array with: (linkTable fieldNamed: 'person_id')))

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').
```

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

Each row in the ADDRESS_BOOK_LINKS represents one entry in an address book. It has has foreign keys to the other tables to indicate which person and which address book the entry is in. It also stores the key for the entry in the dictionary.

```
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).

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.
```

The following shows an example of adding an AddressBook to the database.

```
session inUnitOfWorkDo:
    [addresses := AddressBook title: 'work'.
    addresses
        at: 'musicMan' put: (Person first: 'Sam' last: 'Hinton');
        at: 'author' put: (Person first: 'Martin' last: 'Fowler');
        at: 'self' put: (Person first: 'Roger' last: 'Whitney').
    session register: addresses].
```

Embedded Values[4] – Two Objects One table

There are times when one does not want a separate table for each object. These objects are ones whose identity does not depend on identity[5] like money or date range. In this case we store multiple objects in a single table. As an example we will use a Name object to represent the name of a Person object. First we have an outline of the Person and Name classes. Note that the Name class does not need an id. In the database it is stored in the same row as the rest of the Person object data, which does have a database id.

```
Smalltalk defineClass: #Person
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'name age id '
    classInstanceVariableNames: ''
    imports: ''
    category: 'GlorpExperiments'
```

Class Methods

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

Instance Methods

```
setFirst: firstNameString last: lastNameString
    name := Name first: firstNameString last: lastNameString
```

```
Smalltalk defineClass: #Name
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'first last '
    classInstanceVariableNames: ''
    imports: ''
    category: 'GlorpExperiments'
```

Class Methods

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

Instance Methods

---

[4] See Embedded Value pattern pp 268-271 in Fowler [2003]
[5] See Value Object pattern pp 486-487 in Fowler [2003]

```
setFirst: firstNameString last: lastNameString
    first := firstNameString.
    last := lastNameString
```

The descriptor defines a Person table with four columns. Note the use of the EmbeddedValueOneToOneMapping in descriptorForPerson:.

```
Smalltalk defineClass: #GlorpTutorialDescriptor
    superclass: #{Glorp.DescriptorSystem}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            Glorp.*
            '
    category: 'GlorpExperiments'

allTableNames
    ^#( 'PEOPLE'  )

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

classModelForName: aClassModel
    aClassModel newAttributeNamed: #first.
    aClassModel newAttributeNamed: #last.

classModelForPerson: aClassModel
    aClassModel newAttributeNamed: #id.
    aClassModel newAttributeNamed: #name  type: Name.
    aClassModel newAttributeNamed: #age.

descriptorForName: aDescriptor
    | personTable |
    personTable := self tableNamed: 'PEOPLE'.
    aDescriptor table: personTable.
    (aDescriptor newMapping: DirectMapping) from: #first to: (personTable fieldNamed:
'first_name').
    (aDescriptor newMapping: DirectMapping)  from: #last to: (personTable fieldNamed:
'last_name').
```

```
descriptorForPerson: aDescriptor
    | personTable |
    personTable := self tableNamed: 'PEOPLE'.
    aDescriptor table: personTable.
    (aDescriptor newMapping: DirectMapping)  from: #age to: (personTable fieldNamed: 'age').
    (aDescriptor newMapping: DirectMapping)  from: #id to: (personTable fieldNamed: 'id').
    (aDescriptor newMapping: EmbeddedValueOneToOneMapping)  attributeName: #name.

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)).
    (aTable createFieldNamed: 'age' type: (platform int)) defaultValue: 39.
```

When you read a Person object from the database, as below, the name object for the Person name instance variable is not a proxy. There is no performance reason to read just part of the row, so the name object data is read and the name object is instantiated.

```
result := session readOneOf: Person where: [:each | each name first = 'Roger'].
```

Currently it does not seem possible to use the embedded value as the primary key of the table.

One Object – Multiple Tables

There are times when the data for one object is located in multiple tables. Normalizing a database can be one cause of this. As an example we will use a Person class with the instance variables firstName, lastName and id. The database tables are People and Names with the columns indicated below. The last_name_id column is the foreign key to the Names table.

People

id	first_name	last_name_id

Names

id	last

One could create a class to represent the Names table, but in this example we will only use a Person class. Here is the definition of the class. You can add whatever methods you want to the class.

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

Clear one needs to perform a join on the two tables to get the data we need for a person object. The method descriptorForPerson: contains the description of the join. The rest of the descriptor is straightforward.

```
Smalltalk defineClass: #GlorpTutorialDescriptor
    superclass: #{Glorp.DescriptorSystem}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            Glorp.*
            '
    category: 'GlorpExperiments'

allTableNames
    ^#( 'PEOPLE'  'NAMES' )

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

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

descriptorForPerson: aDescriptor
    | personTable namesTable |
    personTable := self tableNamed: 'PEOPLE'.
    aDescriptor table: personTable.
    namesTable := self tableNamed: 'NAMES'.
    aDescriptor table: namesTable.
    aDescriptor addMultipleTableJoin: (
        Join
            from: (personTable fieldNamed: 'last_name_id')
            to: (namesTable fieldNamed: 'id')).
    (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: (namesTable fieldNamed:
'last').

tableForNAMES: aTable
    (aTable createFieldNamed: 'last' type: (platform varChar: 50)).
    (aTable createFieldNamed: 'id' type: (platform sequence)) bePrimaryKey.

tableForPEOPLE: aTable
    | lastNameId |
    (aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey.
    aTable createFieldNamed: 'first_name' type: (platform varChar: 50).
    lastNameId := aTable createFieldNamed: 'last_name_id' type: platform int4.
    aTable
        addForeignKeyFrom: lastNameId
        to: ((self tableNamed: 'NAMES') fieldNamed: 'id')
```

Association (Tie)Tables – Many-to-many[6]

Association tables, sometimes called tie or link tables, are common. For example consider a database for a bookstore for books on order. We will have table of books and a table of customers. To reuse part of previous examples we will make this a table of people. An individual may have many books on order. Also a given book may be on order by many different people. This is handled by using an association table. Each row of the table can represent the order of a book by one person. The table at least contains a column to refer to a person in the people table and a column to refer to a book in the book table.

[6] See Association Table Mapping pattern pp 248-261 in Fowler [2003]

In the example the association table is called BOOKS_ON_ORDER. To keep the example small the Book class only contains an id and title instance variables. We do not show the standard accessor methods for the Book class. We add an instance variable booksOnOrder to the person class. We show all but the needed accessor methods in the Person class.

```
Smalltalk defineClass: #Book
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'id title '
    classInstanceVariableNames: ''
    imports: ''
    category: 'GlorpExperiments'

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

Class Method

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

Instance Methods

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

addBook: aBook
    booksOnOrder add: aBook

books
    ^booksOnOrder
```

The descriptor has several new things. First of all it has more tables than classes.

```
Smalltalk defineClass: #GlorpTutorialDescriptor
    superclass: #{Glorp.DescriptorSystem}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            Glorp.*
            '
    category: 'GlorpExperiments'

allTableNames
    ^#( 'PEOPLE'  'BOOKS_ON_ORDER' 'BOOKS')

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

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

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

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').
```

The new thing here is the ManyToManyMapping mapping. The amazing thing is that we do not need to include the name of the association table. Glorp will deduce which table is the association table. We will see later how it explicitly list the association table.

```
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: ManyToManyMapping)
            attributeName: #booksOnOrder;
            referenceClass: Book
```

The tables are straightforward. The BOOKS table and PEOPLE table have primary keys and the columns for title and names.

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

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 BOOKS_ON_ORDER table is the association or tie table. Each row represents one book on order for a person. So it has two foreign keys, one to the PEOPLE table and one to the BOOKS table.

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

This is the end of the GlorpTutorialDescriptor class. Here is sample code to add some books and people to the database.

```
session beginUnitOfWork.
books := #( 'Code Complete' 'Palm OS' 'Cat in the Hat' ) collect: [:each | Book title: each ].
session registerAll: books.
session register: (Person first: 'Sam' last: 'Hinton').
session register: (Person first: 'Martin' last: 'Fowler').
session commitUnitOfWork
```

An example of reading books.

```
cat := session readOneOf: Book where: [:each | each  title  = 'Cat in the Hat'].
code := session readOneOf: Book where: [:each | each  title  = 'Code Complete'].
```

Now to have some people order books.

```
session beginUnitOfWork.
sam := session readOneOf: Person where: [:each | each  firstName  = 'Sam'].
sam
    addBook: cat;
    addBook: code.
martin := session readOneOf: Person where: [:each | each  firstName  = 'Martin'].
martin addBook: cat.
session commitUnitOfWork.
```

One can of course read a person and see what books they have on order.

```
sam := session readOneOf: Person where: [:each | each  firstName  = 'Sam'].
sam books
```

A more interesting query is to find all people that have a particular book or books on order. The following query will not work.

```
waitingForCat := session
        readManyOf: Person
        where:
            [:each |
            each booksOnOrder detect: [:book | book title  = 'Cat in the Hat']].
```

While the where block would work if we were dealing with Smalltalk objects. However the where: block is used to generate SQL. There are limited messages that can be sent to the parameter of the where block. One class of messages that can be sent is instance variables of the class. So we can use ‘where: [:each | each booksOnOrder’. Since booksOnOrder is collection we can sent it either anySatisfyJoin: or anySatisfyExists:. We will look at both.

First we have:

```
waitingForCat := session
        readManyOf: Person
        where:
            [:each |
            each booksOnOrder anySatisfyJoin: [:book | book title like: 'Cat%']].
```

This generates the following SQL query on the database.

```
SELECT DISTINCT t1.id, t1.first_name, t1.last_name
 FROM
    ((PEOPLE t1 INNER JOIN BOOKS_ON_ORDER t3 ON (t3.customer_id = t1.id))
        INNER JOIN BOOKS t2 ON (t3.BOOK_ID = t2.id))
 WHERE (t2.title LIKE 'Cat%')
```

A second way is to use:

```
waitingForCode := session readManyOf: Person
            where:
                [:each |
                each booksOnOrder anySatisfyExists: [:book | book title like: 'Code%']].
```

Or equivalently

```
waitingForCode := session readManyOf: Person
            where:
                [:each |
                each booksOnOrder anySatisfy: [:book | book title like: 'Code%']].
```

This results in the following SQL:

```
SELECT t1.id, t1.first_name, t1.last_name
 FROM PEOPLE t1
 WHERE  EXISTS
    (SELECT t2.id
     FROM BOOKS_ON_ORDER s1t1, BOOKS t2
     WHERE
        ((t2.title LIKE 'Code%') AND
          ((s1t1.BOOK_ID = t2.id) AND
            (s1t1.customer_id = t1.id))))
```

You can decide which is better for your situation.

Ordered Reading

A Person object contains a collection of books in the booksOnOrder instance variable. We can specify the order in which this collection is read from the database when we read a Person object. In the following modification of descriptorForPerson: the orderBy: message sent to the ManyToManyMapping orders the books by the title instance variable. Actually the ordering is done on the database side on the column mapped to the title instance variable of the Book class.

```
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: ManyToManyMapping)
        attributeName: #booksOnOrder;
        referenceClass: Book;
        orderBy: #title
```

We can also use a block to indicate the order.

```
    (aDescriptor newMapping: ManyToManyMapping)
        attributeName: #booksOnOrder;
        referenceClass: Book;
        orderBy:  [:each |  each title]
```

While the block syntax is a bit more verbose it allows us to reverse the order as the following illustrates.

```
    (aDescriptor newMapping: ManyToManyMapping)
        attributeName: #booksOnOrder;
        referenceClass: Book;
        orderBy:  [:each |  (each title) descending];
```

It is also possible to explicitly specify the field on the table to used to determine the order.

```
(aDescriptor newMapping: ManyToManyMapping)
        attributeName: #booksOnOrder;
        referenceClass: Book;
        orderBy: [:each |  (each getTable: 'BOOKS') getField: 'title']
```

UseLinkTable

If one looks at this example closely it is not clear how Glorp determines which table is the tie or link table. In the descriptor for the ManyToManyMapping one just provides the name of the instance variable (booksOnOrder) and the type of elements that will be held in the instance variable (Book). If you look at the examples that come with Glorp you will find places where the message “useLinkTable” is sent to the ManyToManyMapping as is done below. This tells Glorp that it needs to find a link table for this mapping. However, since ManyToManyMapping automatically sets the useLinkTable to true, this message is not need except to remind the reader that a link table is used.

```
(aDescriptor newMapping: ManyToManyMapping)
        attributeName: #booksOnOrder;
        referenceClass: Book;
        orderBy: [:each |  (each title) descending];
        useLinkTable
```

Read Write mappings

There are times when the link table is populated via other means and we want to insure that we do not add or deleted link table entries. This can be done by making the mapping read only as is done below.

```
(aDescriptor newMapping: ManyToManyMapping)
        attributeName: #booksOnOrder;
        referenceClass: Book;
        readOnly: true
```

Once this is done we cannot add or delete items from the link table via this mapping. It can be done other ways. However the following code will not add the books to the Sam’s list of books on order. No exceptions are raised, so Sam is added to the PEOPLE table.

```
cat := session readOneOf: Book where: [:each | each  title  = 'Cat in the Hat'].
code := session readOneOf: Book where: [:each | each  title  = 'Code Complete'].
session inUnitOfWorkDo:
    [sam := Person first: 'Sam' last: 'Hinton'.
    session register: sam
    sam
        addBook: cat;
        addBook: code]
```

Cascading

It is common to have nested objects. In the following we have a book inside of the sam object.

```
Sam := Person first: 'Sam' last: 'Hinton'.
catInTheHat := Book title: 'Cat in the Hat'.
sam addBook: catInTheHat.
```

Writing is said to cascade if by writing the outer object the inner object is also written. In GLORP writes are cascaded. So in the following both the Person object and the book object will be written to the database.

```
session inUnitOfWorkDo: [session register: sam]
```

In GLORP updates are also cascaded. If the method fooBar below modifies some instance variable of Sam, the modified instance variable will be updated in the database. It fooBar modifies an instance variable of an instance variable of Sam the database will be updated in the database. So when we make changes to object read from the database committing the changes would update the database regardless of where the object graph the changes occur.

```
session inUnitOfWorkDo:
    [sam := session readOneOf: Person where: [:each | each firstName = 'Sam'].
    sam fooBar.]
```

Deletes do not cascade. So in the code below, deleting sam does not delete the books that sam references. If you want those books to be deleted you need to delete them explicitly. If you set the book references in sam to nil and then delete sam, GLORP will make sure that book links no longer reference the sam row in the database.

```
session inUnitOfWorkDo:
    [sam := session readOneOf: Person where: [:each | each firstName = 'Sam'].
    session delete: sam.]
```

Detached Objects

After reading an object from the database, the object can remain after the session that read it is closed. Such an object is detached from the database.

```
session := codeToGetASession.
[sam := session readOneOf: Person where: [:each | each firstName = 'Sam'].
codeToCloseConnection.
```

There are two dangers in doing this. First keeping an object in memory for long periods of time increases the probability that object's data will not be consistent with the database. Another thread or client may alter the database version and any changes in the in memory object will not be written to the database. Secondly when GLORP reads an object, any references to collections and complex objects are replaced with proxies. The data is not read from the database until the proxy is actually used. The proxy holds a reference to the session used to originally read the object. This session is used to read the collection. If the session has been closed, then an exception is raised when trying to access the collection. So any method in a detached object that references a proxy collection will not work. If the session is not closed (or more accurately the underlying accessor closed), but you no longer have a reference to the session the object is not detached and the proxy will still work.

One can reattach the object to the database by registering it with a session.

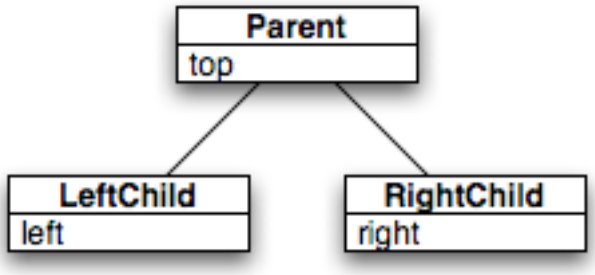
```
session := codeToGetASession.
session register: sam.
```

This should also register all nested objects inside of Sam. It is not clear if changes to sam before you register it will be saved (I have not tried that), but any changes made after you register it will be saved.

There are situations where detached object make sense, but make sure you know what you are doing and understand your situation very well before trying this.

Inheritance and Polymorphic Queries

For our examples we will use the class Parent and two subclasses of Parent, LeftChild and RightChild. Each of the classes has one instance variable as indicated in the following diagram.



A polymorphic query is one were we can query on the Parent class and as a result get instances of Parent, LeftChild and RightChild. If we set up the descriptor to support polymorphic queries then the following query will return instances of all Parent, LeftChild and RightChild objects stored in the database.

```
result := session readManyOf: Parent.
```

There are three common ways to support polymorphic queries with a relational database: table for each concrete class, table for each class and one table for all classes. GLORP currently supports tow. We will look at these two ways.

Table for each Concrete Class[7]

In this method each concrete class in the class hierarchy has its own table. That table holds all the instance variables for the class. For an example of this method we will make the Parent class an abstract class. So we need two tables, which we will call LEFTCHILDTABLE and RIGHTCHILDTABLE. The columns in each table are illustrated below.

LEFTCHILDTABLE

id	top	left_value

RIGHTCHILDTABLE

id	top	right_value

The class definitions are straightforward.

[7] See Concrete Table Inheritance pattern pp 293-301 in Fowler [2003]

```
Smalltalk defineClass: #Parent
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'top id '
    classInstanceVariableNames: ''
    imports: ''
    category: 'GlorpExperiments'

Smalltalk defineClass: #LeftChild
    superclass: #{Smalltalk.Parent}
    indexedType: #none
    private: false
    instanceVariableNames: 'left '
    classInstanceVariableNames: ''
    imports: ''
    category: 'GlorpExperiments'

Smalltalk defineClass: #RightChild
    superclass: #{Smalltalk.Parent}
    indexedType: #none
    private: false
    instanceVariableNames: 'right '
    classInstanceVariableNames: ''
    imports: ''
    category: 'GlorpExperiments'
```

The descriptor uses one new feature a resolver.

```
Smalltalk defineClass: #GlorpTutorialDescriptor
    superclass: #{Glorp.DescriptorSystem}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            Glorp.*
            '
    category: 'GlorpExperiments'

allTableNames
    ^#( 'LEFTCHILDTABLE'  'RIGHTCHILDTABLE' )

constructAllClasses
    ^(super constructAllClasses)
        add: Parent;
        add: LeftChild;
        add: RightChild;
        yourself
```

Note that even though Parent is an abstract class it is still listed in the constructAllClasses method. We do not add a class model for the class.

```
classModelForLeftChild: aClassModel
    aClassModel newAttributeNamed: #left.
    aClassModel newAttributeNamed: #top.
    aClassModel newAttributeNamed: #id.

classModelForRightChild: aClassModel
    aClassModel newAttributeNamed: #right.
    aClassModel newAttributeNamed: #top.
    aClassModel newAttributeNamed: #id.
```

Each of the class descriptor methods register a resolver. The method typeResolverFor: results in a call to typeResolverForParent.

```
descriptorForParent: aDescriptor
    (self typeResolverFor: Parent) register: aDescriptor abstract: true.

descriptorForLeftChild: aDescriptor
    | leftTable |
    leftTable := self tableNamed: 'LEFTCHILDTABLE'.
    aDescriptor table: leftTable.
    (aDescriptor newMapping: DirectMapping) from: #id
        to: (leftTable fieldNamed: 'id').
    (aDescriptor newMapping: DirectMapping) from: #top
        to: (leftTable fieldNamed: 'top').
    (aDescriptor newMapping: DirectMapping) from: #left
        to: (leftTable fieldNamed: 'left_value').
    (self typeResolverFor: Parent) register: aDescriptor.

descriptorForRightChild: aDescriptor
    | rightTable |
    rightTable := self tableNamed: 'RIGHTCHILDTABLE'.
    aDescriptor table: rightTable.
    (aDescriptor newMapping: DirectMapping) from: #id
        to: (rightTable fieldNamed: 'id').
    (aDescriptor newMapping: DirectMapping) from: #top
        to: (rightTable fieldNamed: 'top').
    (aDescriptor newMapping: DirectMapping) from: #right
        to: (rightTable fieldNamed: 'right_value').
    (self typeResolverFor: Parent) register: aDescriptor.

tableForLEFTCHILDTABLE: aTable
    (aTable createFieldNamed: 'left_value' type: (platform varChar: 50)).
    (aTable createFieldNamed: 'top' type: (platform varChar: 50)).
    (aTable createFieldNamed: 'id' type: (platform sequence)) bePrimaryKey.

tableForRIGHTCHILDTABLE: aTable
    (aTable createFieldNamed: 'right_value' type: (platform varChar: 50)).
    (aTable createFieldNamed: 'top' type: (platform varChar: 50)).
    (aTable createFieldNamed: 'id' type: (platform sequence)) bePrimaryKey.

typeResolverForParent
    ^HorizontalTypeResolver forRootClass: Parent.
```

The typeResolverForParent method returns the actual resolver, which is used by GLORP to determine which tables to examine for polymorphic queries. With the above setup the following query will return all Parent, LeftChild and RightChild objects with left as the value of the instance variable top. Note that for the descriptors for LeftChild and RightChild the typeResolverFor:

registers the parent class (Parent), not the LeftChild or RightChild. This is what permits allows the polymorphic query.

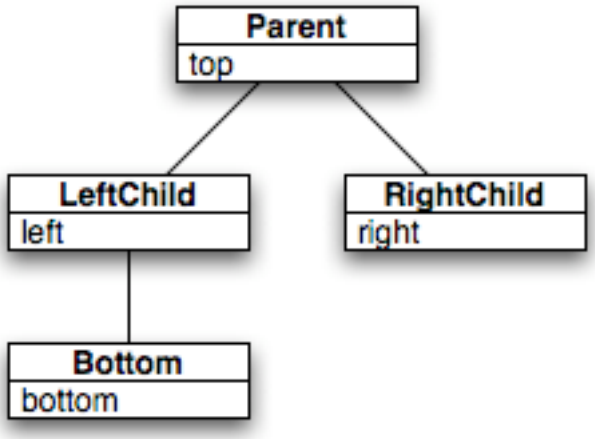
```
result := session readManyOf: Parent  where: [:each | each top = 'left'].
```

We can still write queries directly on the subclasses if appropriate.

```
result := session readManyOf: LeftChild  where: [:each | each top = 'left'].
```

If the inheritance hierarchy is deeper, we may have multiple levels of polymorphic queries. Say we add a Bottom class to LeftChild as shown below. It may occur that we need to write polymorphic queries on Parent's hierarchy and polymorphic queries on LeftChild's hierarchy. If we wish to do this then in the Bottom class descriptor we need the following two type resolvers.

```
(self typeResolverFor: Parent) register: aDescriptor.
(self typeResolverFor: LeftChild) register: aDescriptor.
```



There is one restriction on polymorphic queries when using Table for each Concrete Class, which is one can not use orderBy:. The query makes multiple requests to the database as it has to read both tables, so we cannot have the database order the results for use. As a result the following request will raise an exception.

```
session readManyOf: Parent orderBy: #top
```

One Table for all Classes[8]

In this method of organizing the tables there is one table that holds the data for all objects in the class hierarchy. In this example the one table is called ALL_DATA. The columns of the table are illustrated below. The object_type column is used to store the type of the object in the row. This

[8] See Single Table Inheritance pattern pp 293-301 in Fowler [2003]

allows GLORP to create an instance of the correct class when it reads a row in the table. Note that regardless of what type of object is in a row, one column, either left_value or right_value, will be null. While this method does waste space in the database, performing a polymorphic query will be faster as there is only one table to access.

ALL_DATA

id	top	left_value	right_value	object_type

The class definitions remain the same as in the Table for each Concrete Class example. The descriptor changes a bit. We use a different resolver, need to supply values for object_type column and need more information for the Parent class.

```
Smalltalk defineClass: #GlorpTutorialDescriptor
    superclass: #{Glorp.DescriptorSystem}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: '
            Glorp.*
            '
    category: 'GlorpExperiments'

GlorpTutorialDescriptor methodsFor: 'accessing'

allTableNames
    ^#( 'ALL_DATA')

constructAllClasses
    ^(super constructAllClasses)
        add: Parent;
        add: LeftChild;
        add: RightChild;
        yourself

classModelForLeftChild: aClassModel
    aClassModel newAttributeNamed: #left.
    aClassModel newAttributeNamed: #top.
    aClassModel newAttributeNamed: #id.

classModelForRightChild: aClassModel
    aClassModel newAttributeNamed: #right.
    aClassModel newAttributeNamed: #top.
    aClassModel newAttributeNamed: #id.
```

Note how to indicate the value of the object_type column for LeftChild objects with the resolver.

```
descriptorForLeftChild: aDescriptor
    | leftTable |
    leftTable := self tableNamed: 'ALL_DATA'.
    aDescriptor table: leftTable.
    (aDescriptor newMapping: DirectMapping) from: #id
        to: (leftTable fieldNamed: 'id').
    (aDescriptor newMapping: DirectMapping) from: #top
        to: (leftTable fieldNamed: 'top').
    (aDescriptor newMapping: DirectMapping) from: #left
        to: (leftTable fieldNamed: 'left_value').
    (self typeResolverFor: Parent) register: aDescriptor keyedBy: 'L' field: (leftTable fieldNamed:
'object_type').

descriptorForRightChild: aDescriptor
    | rightTable |
    rightTable := self tableNamed: 'ALL_DATA'.
    aDescriptor table: rightTable.
    (aDescriptor newMapping: DirectMapping) from: #id
        to: (rightTable fieldNamed: 'id').
    (aDescriptor newMapping: DirectMapping) from: #top
        to: (rightTable fieldNamed: 'top').
    (aDescriptor newMapping: DirectMapping) from: #right
        to: (rightTable fieldNamed: 'right_value').
    (self typeResolverFor: Parent) register: aDescriptor keyedBy: 'R' field: (rightTable fieldNamed:
'object_type').
```

Even though Parent is still an abstract class GLORP requires us to given the mappings for the Parent instance variables. If Parent were not abstract we would have to provide a key value for the column object_type.

```
descriptorForParent: aDescriptor
    | table |
    table := self tableNamed: 'ALL_DATA'.
    aDescriptor table: table.
    (aDescriptor newMapping: DirectMapping) from: #id
        to: (table fieldNamed: 'id').
    (aDescriptor newMapping: DirectMapping) from: #top
        to: (table fieldNamed: 'top').
    (self typeResolverFor: Parent) register: aDescriptor abstract: true

tableForALL_DATA: aTable
    (aTable createFieldNamed: 'top' type: (platform varChar: 50)).
    (aTable createFieldNamed: 'left_value' type: (platform varChar: 50)).
    (aTable createFieldNamed: 'right_value' type: (platform varChar: 50)).
    (aTable createFieldNamed: 'object_type' type: (platform varChar: 2)).
    (aTable createFieldNamed: 'id' type: (platform sequence)) bePrimaryKey.

typeResolverForParent
    ^FilteredTypeResolver forRootClass: Parent.
```

Defining the Resolver for a Class

In the above examples the resolver type of the class Parent was defined in a method in the GlorpTutorialDescriptor. If you prefer you can define in the Parent class it self. So you could delete the GlorpTutorialDescriptor>>typeResolverForParent method and add the following class method to Parent

```
Parent class>>glorpTypeResolver
    ^FilteredTypeResolver forRootClass: Parent
```

In Brief
Advanced Reading - Query Objects for Improved Queries[9]

The read methods in a Glorp session are convenience methods for construction query objects. For example the following:

```
employees := session readManyOf: GlorpEmployee where: [:each | each id <= 4].
```

is a convenience for:

```
query := Query readManyOf: GlorpEmployee  where: [:each | each id <= 4].
employees := session execute: query.
```

While the convenience methods are shorter and adequate for most situations dealing with query objects directly give you much more flexibility. The following is not currently possible with convenience methods. The queries results are ordered by name and when names are equal the results are then sorted by id. The ordering is done by the database.

```
query := Query readManyOf: GlorpEmployee where: [:each | each id <= 4].
query orderBy: #name.
query orderBy: #id.
employees := session execute: query.
```

OrderBy: also accepts blocks to indicate the order so we could use query orderBy: [:each | each name] instead of query orderBy: #name.

Query object allow us to read objects in groups. With a session we can read the first N objects that satisfy a query. The following will read the first 50 glorp employees with a salary over 100,000. (We are assuming the salary is an instance variable of GlorpEmployee.) However there is no way using convenience methods to read the rest of the employees that satisfy the condition.

```
employees := session readManyOf: GlorpEmployee  where: [:each | each salary > 100000] limit: 50.
```

The following will read up to 100 employee objects after the first 50 and order them by name.

```
query := Query readManyOf: GlorpEmployee where: [:each | each salary > 100000].
query
    orderBy: #name;
    limit: 100;
    offset: 50.
employees := session execute: query.
```

[9] See Query Object pattern pp 316-321 in Fowler [2003]

When readManyOf: query returns an array of results. You can specify a different collection with the method collectionType:. The following will return a set of employee objects.

```
query := Query readManyOf: GlorpEmployee where: [:each | each salary > 100000].
query collectionType: Set.
employees := session execute: query.
```

See the class SimpleQuery for more methods one can use to refine queries. The class CompoundQuery is useful in joining multiple query objects with Boolean expressions.

Reading Parts of Objects

There are times when you only need part of an object. The following returns just the salaries (assuming salary is an instance variable of GlorpEmployee).

```
query := Query readManyOf: GlorpEmployee where: [:each | each salary > 100000].
query retrieve:  #salary.
salaries := session execute: query.
```

One can retrieve more than one item. In this case for each GlorpEmployee one gets an array containing a salary and a name.

```
query := Query readManyOf: GlorpEmployee where: [:each | each salary > 100000].
query
    retrieve:  #salary;
    retrieve: #name.
salariesAndNames := session execute: query.
```

One can also retrieve a maximum or minimum value.

```
query := Query readManyOf: GlorpEmployee where: [:each | each salary > 100000].
query retrieveMax:  #salary.
maxSalaryInArray := session execute: query.
```

When we know there will be only one value returned we can specify not to wrap the one value in a collection.

```
query := Query readManyOf: GlorpEmployee where: [:each | each salary > 100000].
query
    retrieveMax:  #salary;
    readsOneObject: true.
maxSalary := session execute: query.
```

When using retrieve: the part of the object is selected using SQL in the database, so in general will be more efficient than reading the entire object and selecting the value from the object.

Dynamic Descriptors

The examples in this tutorial treat the GlorpTutorialDescriptor class as static entity. However many of the values set in the GlorpTutorialDescriptor can be changed at runtime. A number of the GLORP test cases illustrate this.

References

Patterns of Enterprise Application Architecture, Martin Fowler, Addison-Wesley, 2003

Document Versions

Date	Description
10/18/2005	Added clarification to polymorphic queries
1/16/2005	Minor corrections, added footnotes & reference, first non-draft version
1/12/2005	Added sections: Direct Instance Variable Access or Accessor Methods; Specifying the Type of Collection; Document Versions Made minor change to Inheritance and Polymorphic Queries
1/11/2005	First public version of document