GLORP

User Guide

By Nevin Pratt

With some updating by Alan Knight

Document Introduction

While GLORP does not yet have any formal documentation, Nevin Pratt, maintainer of the Squeak port, has written the following documentation in conjunction with the original Squeak port of GLORP. I've updated it slightly to reflect recent changes in Glorp and we are making it available as part of the standard Glorp distribution. Note that this should not be considered as formal documentation, is not in any way connected with the regular VisualWorks documentation, and does not necessarily reflect the views of Cincom or anyone else except Nevin. I'd also suggest skipping over the level zero and 1 strategies and going to the more recommended approaches that start with the "Basic Glorp Concepts" section. However, given all those disclaimers, I think it should be very helpful
-- Alan Knight

GLORP Introduction

GLORP stands for *Generalized Lightweight Object-Relational Persistence*. It is a framework for giving your domain objects the ability to persist beyond your current program invocation. It also, as a side-effect, enables your program to have *multi-user* abilities.

To enable your domain objects to persist, theoretically all you need is some place (any place) to store bits. GLORP uses a relational database (RDBMS) for this. Currently (version 0.3) the supported databases include Oracle, Postgresql, SQL Server, Ocelot, and MS-Access, and it's fairly easy to add a a database as long as the basic connection is available. Partial support for some other databases is also available.

GLORP is relatively platform-independent, running on many different operating systems with ports available for many different Smalltalk dialects, although there is a small amount of well-isolated, platform-dependent code for each port. It is also the framework that Cincom (the vendor for VisualWorks) anticipates using as the core mapping layer for their next generation of database frameworks.

The goal of GLORP is to make your RDBMS (relational database) transparently appear to be what many in the industry refer to as an *Object Database*. As such, you theoretically don't have to know any SQL to use it, but what you'll discover is that while the illusion is very good, it isn't perfect (and probably can't be perfect, given the constraints of the RDBMS world), and thus it helps to know a little SQL.

But even so, the approach used by GLORP is light-years beyond the typical approach used for other so-called "frameworks" for other languages, as those other frameworks typically just give you a communications pipe to the database, plus a place to write your own SQL code for reading/writing the database. That approach is what I call *Direct SQL Coding*. I also refer to that approach in this guide as a "Level 0" approach, with the "zero" intended to represent that such an approach is worth (almost) zero, or nothing. Thus, *Direct SQL Coding* is almost worthless as a persistence technique in a large program. And yes, GLORP certainly supports *Direct SQL Coding*, if that is your desire, and I even show you how. But your aim should be higher than that, and I also show you how to aim higher.

The main GLORP web-site is at http://www.glorp.org, and that site also has some documentation. Note, however, that the main person behind that web site is Alan Knight, and Alan now works for

Cincom. Thus, the newer GLORP builds tend to appear for VisualWorks (VW) first, and tend to appear on the *Cincom Public Repository* for VisualWorks before they end up on the glorp web site.

GLORP Strategies – Level 0
Direct SQL Coding

The simplest (as well as least extensible and usually least maintainable) relational storage strategy is to embed SQL directly into code whenever an object is read or written. With this approach, you explicitly code all of the table operations in SQL by hand, and embed the SQL code directly into your application code. This is the basic approach used by most of the various Java code I've been unfortunate enough to have had to look at (one exception being the slightly improved approach afforded by the Java "Struts" framework, and another exception another slightly improved approach using *Enterprise Beans)*. If you use *Direct SQL Coding* with GLORP, you will bypass all of the Object-Oriented benefits offered by GLORP, just as you lose all of the OO benefits using this approach with Java.

This approach is relatively easy to visualize and initially implement, and for extremely simple applications, it might even be an acceptable coding strategy. It should be noted, however, that if you are going to embed SQL directly into the code like this, there is really no need for GLORP at all, because all you need to support this style is a way to send SQL directly to the RDBMS. And of course, the EXDI layer (for VisualWorks) can do that, as can the PostgreSQL driver for Squeak[1]. For that matter, any RDBMS database driver can send SQL directly to the RDBMS—otherwise it's not much of a database driver, is it!?

Never-the-less, for understanding parts of GLORP, it is probably useful to know how you would do this using GLORP. And so here are the details.

It is an instance of a platform-specific subclass of the *DatabaseAccessor* class that allows you to issue direct SQL code from your application to your RDBMS[2]. However, while the *DatabaseAccessor* subclass is platform-specific, it is instantiated in a platform-neutral manner, through the *DatabaseAccessor* class itself. Thus, you don't need to know which subclass of *DatabaseAccessor* to instantiate. You instead just hand the *DatabaseAccessor* class an instance of a *Login* (which is just a data structure containing login parameters), and it automatically selects the appropriate subclass to instantiate. In the example below, we create a login instance for accessing the 'db' PostgreSQL database residing on the machine named 'host', and with user name of 'username' and password of 'password':

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

The 'database' login attribute for the Login instance can be an instance of any of the subclasses of *DatabasePlatform*. These classes each contain some database-specific information needed for GLORP to function properly on those respective platforms, but unless you are wanting to extend GLORP to support other databases, you probably don't need to otherwise know anything about those two classes. Furthermore, those classes employ various helper classes, primarily under the DatabaseSequence and

1The Java analog is typically JDBC, but as I mentioned, occasionally *Struts* is added to the mix. Java code rarely progresses beyond that.

2It basically does this by passing the SQL directly through to the database layer, such as the EXDI layer for VW, or the PostgreSQL driver for Squeak. So again, this is an indication that you really don't need GLORP if this is all you are using for.

DatabaseType hierarchies, which you also probably don't need to learn anything about, but are mentioned below just so you know which classes you can ignore.

Once you have created your *Login* instance, you create a platform-specific *DatabaseAccessor* subclass instance thus:

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

Now, for Squeak, your 'accessor' from above will automatically end up being an instance of *SqueakDatabaseAccessor*. For VisualWorks, it will automatically be an instance of *VWDatabaseAccessor*. For Dolphin, it will automatically be an instance of *DolphinDatabaseAccessor*, and for VisualAge, it will automatically be an instance of *VA55DatabaseAccessor*.

Once you've gotten your 'accessor', you can tell it to log in to your database:

```
accessor login.
```

Once you've logged in, you can tell it to begin a transaction:

```
accessor beginTransaction.
```

Then you can tell it to execute any arbitrary SQL string. The result will be an array of values representing the rows returned, if any:

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

Once you have issued one or more SQL statement(s) within a transaction, you can tell it to either commit or rollback the transaction:

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

And you can tell it to log out of the database:

```
accessor logout.
```

If the string passed to #executeSQLString: contains invalid SQL, an exception will be triggered. Therefore, you might want to wrap that call in an exception handler. Or you can use the following simple method that wraps it for you, which is functionally identical to wrapping it yourself:

```
accessor doCommand: [result := accessor executeSQLString: aSQLString]
         ifError: errorBlock
```

In fact, the above is absolutely identical to the following:

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

Since those two are identical, and the latter is just as simple as the former, it leads one to wonder why the first even exists, because it arguably doesn't seem to improve anything. So why was #doCommand:ifError: even created?

The only reason I can think of is to provide for the possibility of differing Smalltalk implementations having different ways to create exception handlers. For example, at one time VisualWorks used #handle:do:, while VisualAge used #on:do:. However, the #on:do: form was adopted by the ANSI standard, and I believe every Smalltalk dialect now implements it.

There is potentially one more advantage to using #doCommand:ifError: rather than #on:do:, and that is you can easily search for senders of #doCommand:ifError: to find all exception handlers within the GLORP framework separately from the rest of the image.

Anyway, both forms work fine—use whichever you prefer. I personally prefer #doCommand:ifError:, and it's partially under the belief that the original author had something more in mind for that method.

Transaction Boundaries

So why should we do this:

```
accessor beginTransaction.
```

rather than this:

```
accessor executeSQLString: 'BEGIN TRANSACTION'.?
```

And likewise, why should we send #commitTransaction or #rollbackTransaction to the database accessor rather than issuing those respective SQL statements directly?

The answer is that #beginTransaction, #commitTransaction, and #rollbackTransaction all defer to the *connection* object, relying on it to do the platform-specific "right" thing to mark the transaction (of course, that is all that #executeSQLString: does as well, is defer to the *connection* object).

In the case of Squeak (using PostgreSQL), the *connection* object is an instance of *PGConnection*, and sending #beginTransaction to the accessor just makes the connection object send 'BEGIN TRANSACTION' to the RDBMS, exactly the same way that would have happened had you just sent 'BEGIN TRANSACTION' explicitly via #executeSQLString:. So, for Squeak, there really isn't a difference in the above forms.

But for some Smalltalk dialect and RDBMS combinations, there seems to be a difference. Hence, if you mark your transaction via 'accessor beginTransaction' rather than using #executeSQLString:, you can potentially keep your code more portable.

But I'm not sure that it ever matters. Specifically, for Squeak, there is no difference at all in whether you tell the database accessor to #beginTransaction, or if you send the SQL string 'BEGIN TRANSACTION' directly to the database yourself via the #executeSQLString: method. No difference at all.

So, again, take your choice. But I again recommend using the #beginTransaction, #commitTransaction, and #rollbackTransaction methods for consistency.

GLORP Strategies – Level 1
Direct SQL Coding – Without the SQL

SQL is not very difficult, and in fact was a distinct improvement for certain cases when it was invented. That is because SQL is designed to work with complete sets of data all at once, whereas the predominant technology at the time SQL was invented was procedural, and you were forced to deal with the data one row at a time procedurally.

But what if I don't know SQL? Can I still use GLORP to explicitly code table operations by hand?

Yes, using this *Without the SQL* variation of the *Direct SQL Coding* strategy. This variation is related to the *Direct SQL Coding* strategy (and hence, shares its name), because with *Direct SQL Coding* you are explicitly coding table operations by hand. And, just like the previously section described, the *Without The SQL* variation also doesn't offer any real benefits in terms of Object-Oriented capabilities.

This variation also happens to be the approach used by many so-called *Object-Oriented RDBMS Interfaces* for other languages. They merely provide a way for you to read, write, or update rows in the database, and leave it to you to figure out what to do beyond that. They typically do this by providing a *Table* object of some kind, a *Column* object of some kind, and a *Row* object of some kind, and these three types of objects are directly mapped to the table, column, and row concept of a relational database. Thus they often trade (relatively) clean SQL syntax for a more complicated object API to do exactly the same thing, and then call it object-oriented (when it really isn't) just because they happen to have the tables, columns, and rows representable as objects now. NeXT's long-defunct DB-Kit was one such beast, and there are (and were) others[3]. Beware of such systems, as they often turn out to be more complicated than if you had just directly coded the SQL yourself, using the earlier *Direct SQL Coding* strategy.

With a relational database, you obviously first want to login to the database (as well as logout later). After that, there are essentially six additional things that you will want to do with a table[4] (which means that there are seven things total):

1. Login/Logout.
2. Create a table.
3. Delete a table.
4. Insert a row.
5. Delete a row.
6. Update a row.
7. Select a row.

This section teaches you how to do each of those seven things, and without knowing any SQL at all to do them. In many cases, though, I think it is more complicated doing these things without SQL than if you just directly coded the SQL yourself, but the GLORP layer that allows you to do this is a low-level

3*Java Enterprise Beans* code that you'll find at a typical company also generally uses this approach. Although the beans theoretically could use a more OO approach, in typical real-world code, as well as typical Java literature, you'll find that it does not.

4This ommits other potential things like creating indexes, etc. These other things are typically DBA tasks.

layer upon which the rest of GLORP depends, and it is therefore quite useful to know these details (and of course, this represents an area where GLORP, as well as any other mapping product, fails to deliver *The GemStone Illusion*, which is the illusion of an ODBMS that I describe in the next section).

Login/Logout:

Here is how you log in. And as should be obvious, you need to change the *username* and *password* arguments with something that is appropriate for your installation, and you need to also change the hostname (eg, 'host') and database name (eg, 'db') in the connect string to something that is appropriate for your installation.

```
login := Login new database: PostgreSQLPlatform new;
   username: 'username';
   password: 'password';
   connectString: 'host' , '_' , 'db'.
accessor := DatabaseAccessor forLogin: login.
accessor login.
accessor logout.
```

The database accessor created during login is used throughout the code that is subsequently shown below, so anytime you see *accessor* in any of the code that follows, you now know where it came from.

Create A Table:

As a specific example of creating a table, consider a table that has two columns: a 'cust_no' column containing an integer, and a 'cust_name' column containing up to 255 characters. Furthermore, consider that the 'cust_no' is the key for each record. If you were using SQL, you might declare the table from a *psql* session[5] thus:

```
create table my_customer(
    cust_no     integer CONSTRAINT my_customer_PK PRIMARY KEY,
    cust_name   varchar(255));
```

As a brief note about the above SQL statement, one common convention for RDBMS systems is to have SQL keywords in uppercase, and other names (such as table names and column names) as lowercase, and to separate the words within a name with an underscore. This is a different convention than we are used to with Smalltalk, but you need to realize that the RDBMS world is a different world than the Smalltalk world, and you will probably need to conform to the conventions that non-Smalltalk people (such as possibly a DBA) have already imposed. GLORP tries to be flexible in this regard, and allow any convention, but there are a few areas where GLORP isn't (yet) flexible. For example, GLORP wants the names (table names, column names, constraint names, etc.) to begin with a letter, and then you can use any combination of letters, numbers, or underscores, but you cannot *begin* a name with an underscore[6].

5Refer to the PostgreSQL documentation for information about their *psql* tool and how to use it.

6It is actually a Squeak Smalltalk limitation that imposes this constraint, not GLORP. In fact, earlier versions of GLORP allowed the leading underscores. But,since GLORP is trying to be portable across Smalltalk implementations, GLORP now honors that constraint, and you may not have leading underscores in the names.

Furthermore, constraint names must have the same name as the table name, and then appended with particular suffix's. For example, notice that for the above, the primary key has been given the same name as the table name, but is suffixed with '_PK'. This particular convention is currently mandated by GLORP[7].

Now, instead of creating the above *my_customer* table from within *psql* as that SQL showed, we really wanted GLORP to do it. So if you executed the above from a *psql* session, then let's drop the table (again from a *psql* session) now:

```
      drop table my_customer;
```

To create this table from within GLORP (instead of *psql)*, you create an instance of *DatabaseTable* that describes the table that you want:

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

You then hand that table instance back to the *database accessor* that you created during *Login*, telling it to create the table, and the database accessor does the job:

```
     accessor createTable: table ifError: [].
```

The above statement will autocommit. But, if you prefer, you could wrap the above in an explicit transaction thus:

```
      accessor beginTransaction.
      (accessor createTable: table ifError: []) notNil
              ifTrue: [accessor commitTransaction]
              ifFalse: [accessor rollbackTransaction].
```

So, doing all of the above as a single "do-it" operation within a Squeak workspace, on the localhost machine, using database 'bb', username 'postgres', and no password, (and using autocommit), you can create the table indicated by doing the following:

7See `DatabaseTable>>primaryKeyConstraintName` for a hint of why this is so.

```
| login accessor table keyField |
login := Login new database: PostgreSQLPlatform new;
        username: 'postgres';
        password: nil;
        connectString: 'localhost' , '_' , 'bb'.
accessor := DatabaseAccessor forLogin: login.

table := DatabaseTable named: 'my_customer'.
keyField := table createFieldNamed: 'cust_no' type: accessor platform int4.
table addAsPrimaryKeyField: keyField.
table createFieldNamed: 'cust_name'
        type: (accessor platform varChar: 255).

accessor login.
accessor createTable: table ifError: [].
accessor logout.
```

Delete A Table:

Deleting a table is similar to creating a table, in that you hand that table instance back to the *database accessor* that you created during *Login*, telling it to delete the table, and the database accessor does the job:

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

And of course again the above statement will autocommit. And again, if you prefer, you could wrap it in an explicit transaction thus:

```
accessor beginTransaction.
(accessor dropTable: table ifAbsent: []) notNil
       ifTrue: [accessor commitTransaction]
       ifFalse: [accessor rollbackTransaction].
```

There are also easier ways to drop the table, only requiring you to know the table name:

```
accessor dropTableNamed: aString.
```

But this form of dropping the table doesn't trap for the error of the table not existing. To do that, you can use the following form:

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

And again, you could wrap any of the above in an explicit transaction rather than depending on autocommit.

So, doing all of the above as a single “do-it” operation within a Squeak workspace, on the localhost machine, using database 'bb', username 'postgres', and no password, (and using autocommit), you can delete the table indicated by doing the following:

```
| login accessor table keyField |
login := Login new database: PostgreSQLPlatform new;
        username: 'postgres';
        password: nil;
        connectString: 'localhost' , '_' , 'bb'.
accessor := DatabaseAccessor forLogin: login.
accessor login.
accessor dropTableNamed: 'my_customer' ifAbsent: [].
accessor logout.
```

Insert A Row:

To insert a row without explicitly using SQL, we create an instance of DatabaseRow from the *table* in the code above:

```
row := DatabaseRow newForTable: table.
row at: (table fieldNamed: 'cust_no') put: 3.
row at: (table fieldNamed: 'cust_name') put: 'Donald Duck'.
```

We can then create a command that knows how to insert this row.

```
command := InsertCommand forRow: row useBinding: true platform: login
database.
```

Normally, however, you wouldn't do this sort of operation independently, but rather through the *GlorpSession* class. GLORP is designed so that the bulk of all activity happens around a session. The session normally decides whether writing a row for a given *DatabaseRow* should involve an *insert, an update*, or a *delete*, and it can generate a proper SQL string for any of those possibilities. Thus, the session has the responsibility of generating that SQL. But I am bypassing all of that machinery, and just asking the session to give me a proper SQL INSERT operation, because I am taking the responsibility myself to make sure that I indeed need an INSERT rather than, say, an UPDATE. Note that the useBinding flag tells the system whether to use database binding against the prepared statement in order to put in the row data, or to generate a SQL string with the data directly embedded. Binding is usually preferable, although not necessarily supported by GLORP on all Smalltalk dialects. Using binding also allows the server to potentially re-use prepared statements, which can be an important optimization.

After getting the INSERT command from the session, you then tell the database accessor to execute it:

```
accessor executeCommand: command.
```

(Note that, unfortunately, the doCommand: and executeCommand: operations sound very similar, but aren't related)
Thus, the complete code for doing all of the above as a single “do-it” operation within a Squeak workspace, on the localhost machine, using database 'bb', username 'postgres', and no password, (and using autocommit), becomes:

```
| login accessor table keyField row sqlString command |
login := Login new database: PostgreSQLPlatform new;
        username: 'postgres';
        password: nil;
        connectString: 'localhost' , '_' , 'bb'.
accessor := DatabaseAccessor forLogin: login.

table := DatabaseTable named: 'my_customer'.
keyField := table createFieldNamed: 'cust_no' type: accessor platform int4.
table addAsPrimaryKeyField: keyField.
table createFieldNamed: 'cust_name'
        type: (accessor platform varChar: 255).

row := DatabaseRow newForTable: table.
row at: (table fieldNamed: 'cust_no') put: 3.
row at: (table fieldNamed: 'cust_name') put: 'Donald Duck'.

command := InsertCommand forRow: row useBinding: true platform: login
database.

accessor login.
accessor executeCommand: command.
accessor logout.
```

Delete A Row:

Deleting a row is identical to inserting a row, except that the we generate a delete command instead of an insert command.

```
command := DeleteCommand forRow: row useBinding: true platform: login
database.
```

Update A Row:

Updating a row is also identical to inserting a row, except that the session is asked for an update command for the row instead of the INSERT string. Also, since this is an update, you presumably have something you want to update, or change. I will change the customer name in the example below:

```
row := DatabaseRow newForTable: table.
row at: (table fieldNamed: 'cust_no') put: 3.
row at: (table fieldNamed: 'cust_name') put: 'Mickey Mouse'.

command := UpdateCommand forRow: row useBinding: true platform: login
```

```
database.

accessor login.
accessor executeCommand: command.
accessor logout.
```

The above code snippet will cause the following SQL to be generated and executed in the RDBMS:

```
UPDATE my_customer
SET cust_no = 3, cust_name = 'Mickey Mouse'
WHERE cust_no = 3
```

Select A Row:

The seventh and final thing to show is how to select a row without using SQL to do it. Unfortunately, there is no easy way to get GLORP to generate a SELECT statement without delving deep into descriptors and mappings, neither of which have been introduced to you yet (but both of which will be talked about later).

For now, if you really want to bypass all of the automatic facilities provided by GLORP with your own explicit SELECT statements, and if you also don't want to write SQL, then I recommend that you generate the DELETE statement, (via the section on *Delete A Row*), ask it for its #sqlString and then replace the leading 'DELETE' within that string with 'SELECT *':

```
command := DeleteCommand forRow: row useBinding: true platform: aPlatform.
sqlString := command sqlString.
sqlString := 'SELECT *', (sqlString allButFirst: 6).
```

Of course, the above means that you are in fact writing a fragment of SQL yourself (by replacing DELETE with SELECT *), so I didn't completely show you how to avoid writing SQL for the SELECT case. *(Note: I would really, really, really recommend against doing this, and going on to further sections to find out about using descriptors and mappings -- Alan Knight)*

At some later time when you know about descriptors and mappings, selecting from the database will be reduced to something like the following:

```
myCustomer := session
                  readOneOf: MyCustomer
                  where: [:each | each custNo = 3].
```

There are also a number of specialized query classes that can aid you in selecting. This will be covered later.

Basic GLORP Concepts

This section covers basic GLORP concepts that you need to be aware of before you move on to the rest of this guide. Although I call it *Basic GLORP Concepts,* the concepts covered here utilize more advanced features of GLORP than the prior sections utilize. In particular, pay attention to the details leading up to section on *The GemStone Illusion*, because this describes the "target", or ideal, that GLORP is shooting for.

No Explicit Write Operations:

There is no explicit *write* operation of any form with GLORP (or at least, not when using the automatic facilities, but you can explicitly write if you want to, just as the preceding sections showed).

In the preceding sections, you learned how to utilize the lowest levels of GLORP to directly manipulate tables, rows, and columns the RDBMS. You should know that such an approach (of direct table manipulation) goes against the grain of the basic design ideas behind GLORP.

When you are using the higher-level functions of GLORP, it will watch for *dirty objects* (objects whose state has been changed since the transaction began) and then automatically write those objects to the RDBMS when you commit the transaction (actually, when you commit the "Unit Of Work", but that's a slightly more advanced topic). It automatically generates the required SQL necessary to do that. You don't (or shouldn't) need to ever explicitly write any SQL yourself, or otherwise need to explicitly manipulate the rows yourself.

So, how does it know how to write those objects to the RDBMS? How does it know which tables, columns, and rows to create or update, and when?

It knows through the declarative mappings you create that describe the relationships between the tables/columns and the objects. Such mappings are known as *meta-data* to GLORP.

Meta-data Driven:

The meta-data is a declarative description of the correspondence between an object and its database representation. It is used to drive reads, writes, joins, and anything else that is needed. SQL code should not ever have to be explicitly created, as it is autogenerated through the mappings, (i.e., through the meta-data).

To map, we need a model of the table structure, and a model of the object structure. Currently both are built up in code. We expect that in the longer term, table structure might be imported from the database. There might also be some mechanism for more easily describing the mapping, for instance graphical mapping tools But for now, the mappings must be built via explicit code. But don't worry—it's not that hard!

Object Identity:

Preserving *object identity* is one of the failings of the *Direct SQL Coding* approach previoiusly mentioned. For example, suppose you needed the instance of an *Individual* known as “Nevin Pratt”, and you were to directly code a database fetch to get the data for the “Nevin Pratt” individual. After explicitly fetching the data, you would then typically instantiate an *Individual* and set the internal data of that individual to the values that you explicitly fetched, via something like:

```
nevin := Individual new.
nevin firstName: fetchedFirstName.
nevin lastName: fetchedLastName.
      ...etc...
```

This is how you would “materialize” the “Nevin Pratt” instance into a complete instance from the data in the database, using the *Direct SQL Coding* approach.

But now what if somewhere else in your program you also needed to reference the “Nevin Pratt” instance? Using the *Direct SQL Coding* approach, that other code would likely again fetch the data, then would likely instantiate yet another *Individual* instance, and then would likely set the internal data of that new instance to the values that were just fetched.

But now you have two “Nevin Pratt” objects when there really should only be one. This is obviously not good.

So, how do you avoid making two instances?

You avoid it by preserving *object identity*. To preserve object identity, the system needs to be smart enough to recognize when the object has already been materialized, so that the second time it is asked for the object, it merely returns the original object instead of re-materializing it a second time.

To preserve object identity, the framework itself needs to be given the responsibility of instantiating the instance, so that it can make a decision as to whether or not it actually needs to instantiate it again. This all by itself disallows the use of the *Direct SQL Coding* approach. We've got to go with something more intelligent than that.

Furthermore, preserving object identity absolutely requires that we employ an object cache of some kind, so that the cache can be queried to see if the desired object is already in cache, or if it needs to be brought in from disk for the first time.

You may have already figured out from the discussion in the *No Explicit Write Operations* and *Meta-data Driven* sections that GLORP manages the instance creation itself, but you might not have thought about or otherwise been aware of the object cache.

Yes, GLORP manages it's own object cache. And with that, GLORP also preserves object identity. In fact, identity is preserved at all levels of the system, from the internal API's to the public API's. Thus, asking for the same descriptor twice will produce the identical descriptor. Asking for the same table twice will produce the identical table, and fields within the table are treated likewise internally.

Preserving identity uniformly on these objects is fundamental to the operation of GLORP. Not only does this allow more efficient lookups, but it avoids issues of name matching, complex printing, copying, and so forth.

Proxied:

GLORP uses proxies in it's object cache. These are objects that stand in for the real object, but can be transparently replaced by the real object at a later time. Using proxies speeds up many operations because GLORP doesn't have to instantiate all the domain objects that can be reached from a given root domain object all at once. Also, it reduces memory demands because a proxy takes up much less memory than the real object does. For the most part, the use of proxies is transparent to the programmer.

Non-Intrusive:

Suppose your application was a simple, single-user, single-machine application, and you believed it would never grow beyond that? For such a simple application, the easiest persistence strategy (at least for Smalltalk) would be to just use the *image save* functionality that is built-in to almost every Smalltalk implementation. You would not need to use an RDBMS for persistence. You would not need to use extra external files for persistence. You would not need to do anything extra for such persistence—right?

Of course, you would still want to separate the domain behavior from the presentation layer (typically the "GUI"), and create separate objects in separate layers for this (you would do this for reasons that are beyond the scope of this GLORP guide). But you really wouldn't need to do anything special for persistence.

Now, suppose you wrote such an application, believing it would never need to grow beyond this simple model, but then later you discovered that you really *did* need to grow it. Are you, as they say, SOL (hmm, *Surely Outa Luck?*)?

Not at all—at least, not if you later add a *non-intrusive* persistence layer. What "non-intrusive" means, in this context, is that it should be possible to use most of the facilities of your persistence framework (ala GLORP) for your domain layer objects, and do it without having to specially modify the objects in your domain layer to accommodate adding the persistence framework.

Furthermore, if you wish to utilize an existing RDBMS schema for your "persistence store", *non-intrusive* means that it should be possible to map your domain objects to an existing schema without having to modify the schema to accommodate the domain objects.

In short, *non-intrusive* means that you can map an arbitrary domain model to an arbitrary relational model.

A truly non-intrusive persistence framework would therefore make persistence issues become a deployment issue rather than a development issue, because you could develop within the simple single-user model previously described, and yet choose to later deploy under a multi-user model later, and do it with no impact to development *at all.*

Can GLORP do that?

Well, no, nothing can really do that, and no RDBMS mapping tool ever will. Nothing will ever be able to map arbitrary models that way.

But GLORP actually comes pretty close—closer than any other RDBMS persistence framework I've looked at[8]. GLORP can accommodate a wide variety of database schemas and object models, in a very non-intrusive manner.

And this actually makes it feasable to defer many persistence decisions to deployment time rather than development time.

It is the design of the descriptor subsystem within GLORP that achieves this non-intrusiveness. In other words, to use GLORP terminology, it is the *mappings* that do it.

Of course, you may have already figured that out after reading the previous section about GLORP being *Meta-data Driven*. The meta-data *is (are)* the mappings.

The GemStone Illusion:

If GLORP could truly be 100% non-intrusive to the domain model as described above, then it would automatically be able to provide what I call *The GemStone Illusion*. Thus it would provide the illusion of having an active object database like GemStone for your persistence rather than using a relational database.

In GemStone, when it is used as an Application Server (meaning that you have written your application in GemStone Smalltalk, and are running the application from within GemStone) persistence automatically happens, with no extra effort on your part. Unfortunately, I've seen many GemStone installations that used GemStone more as a database than as an Application Server. When used as a database, you move data from GemStone into your client application, process it there, and then move it back to GemStone afterwards. When GemStone is used as a database, it is not at all transparant, and such a use definitely impacts your domain layer. But if your application instead runs completely from within GemStone, you can give your domain model automatic, transparent persistence, with no added effort, and potentially with no impact on your domain layer design.

Note that *The GemStone Illusion* doesn't require that you need to be able to utilize an arbitrary RDBMS schema (because GemStone isn't an RDBMS). *The GemStone Illusion*, as I am defining it, only requires that you be able to map persistence to an RDBMS with zero impact on the domain layer design, such as what you would have if you instead implemented the domain layer completely within GemStone.

And of course, such a capability would in fact push most persistence issues into being just a deployment

8Note that this statement explicitly omits *Object Application Server Technologies* (commonly called an ODBMS) from the discussion, as they have no intrinsic need for such a persistence framework.

issue instead of a development issue.

If you can come fairly close to achieving *The GemStone Illusion*, it enables you to write your application without regard to database issues, and in fact you can first develop a completely functional single-user program with no database what-so-ever[9]. You can then later just tack the database of your choice on afterwards to make the program multi-user (either that, or else you can just go ahead and use GemStone at that point in time instead of trying to create an illusion of using GemStone[10]).

9In this configuration, the standard Smalltalk image save capability is probably sufficient for persistence.
10Assuming you used GemStone as an Application Server rather than as a database.

GLORP Strategies – Level 2

Classes As Storage Managers

Class-Side Methods:

A slightly better variation of the *Direct SQL Coding* strategy (either with or without explicit SQL) is to embed class-side methods within each domain class that can be used to explicitly read or write instances of that class to or from the RDBMS. This is how *GemConnect* for GemStone works (GemConnect allows GemStone to talk to RDBMS's). Since GemStone really doesn't need to use the RDBMS for persistence, GemStone typically doesn't need the level of sophistication that a product like GLORP provides, as it often just needs a simple way to read and write rows to the RDBMS.

But the only real difference between this approach and the *Direct SQL Coding* strategy is that this approach at least attempts to centralize the SQL to a single location for each class. Other than that, this approach also tends to lead to non-Object-Oriented persistence designs, just as *Direct SQL Coding* does.

With such an approach, persistence is not transparant because each object that needs to be persisted has to be explicitly written to the database, and each object that needs to be materialized from the database has to be explicitly read from the database. This approach also typically results in a proliferation of query methods on the class side of each domain class to support the various queries that are explicitly asked for via the rest of the application code.

In the case of GemConnect, there is a bit more going on, because GemConnect also employs various caches. GemConnect also supports the *Without the SQL* variation of the *Direct SQL Coding* approach, as it has various classes that represent tables, columns, and rows, and can generate SQL for you for explicitly reading or writing rows.

But GemConnect fails to provide *The GemStone Illusion* in any meaningful way. But then again, it doesn't try to do that, nor does it need to do that, for the simple reason that it already employs GemStone, and thus doesn't need to provide an *illusion* of GemStone. What would be the point of the illusion?

Tables as Classes, Rows as Instances

When using *Classes As Storage Managers* strategy, a typical design technique is to create a class for each table. The columns of the table would be for the instance variables of the class, and the rows of the table would represent the instances of the class. Thus, we have:

Table	=	Classes
Column	=	Instance Variable
Row	=	Instance

I have seen many persistence frameworks take such a simplistic approach, including most (all?) Java code I've looked at. Such an approach has a myriad of problems, but all the problems seem to revolve around this fundamental difference:

> Classes are repositories for behavior
> but
> Tables are repositories for data

As repositories of data, tables are usually designed to fit strict rules, known as *Rules of Normalization,* or *Normal Forms*. Tables are thus optimized for *select, insert, delete,* and *update* operations, which I will collectively refer to as *Query Operations*. In short, tables are optimized for queries.

In contrast, classes have no particular basic need to conform to normal forms. The primary consideration in their design should be upon the behavior that they support or perform, and the internal instance data of the class is often merely a side-effect of supporting that behavior.

That's a pretty dramatic difference in design focus.

Furthermore, simplistically mapping tables directly to classes does not take into account object *inheritance,* where instance variables are shared commonly with a common superclass. For example, suppose you have an object model with a *LegalEntity* abstract superclass, and *Company* and *Individual* subclasses (and possibly *Government* too). And under those you might have various specializations of companies and individuals (and maybe governments). The *Company* class might be used to represent any company that does not satisfy any of the specialization subclasses. Thus, any given specialization subclass of *Company* is a concrete class, and it would have a concrete superclass *Company*, which in turn would have an abstract superclass *LegalEntity*. And each of these classes would have it's own instance variables defined (just as *Individual* and *Government* subclasses of *LegalEntity* also would).

Now, if classes are tables, then you would have to map the *Company* class to it's own table, you would map subclasses of *Company* to their own tables, you would map the *Individual* class to it's own table, and you would map the *Government* class to it's own table. And each of these tables would have to carry the complete set of instance variables that a fully instantiated and initialized class instance would hold, without regard to inheritance.

To further complicate things, your friendly neighborhood DBA would then see all those tables, and notice the "seemingly" common datum elements being duplicated among the tables, and would instinctively push to normalize the tables. And he would probably win over the opinion of management, and succeed in normalizing the tables, and thus succeed in breaking your simplistic object mapping scheme in the process. And you wouldn't be able to do anything about it.

You really need a better answer than that.

Never-the-less, GLORP can certainly be used with this strategy, even though I would rate this strategy as only slightly better than the *Direct SQL Coding* strategy.

Code examples for the *Classes As Storage Managers* strategy will be left as an exercise for the reader. Just remember, though, it is the *Direct SQL Coding* strategy (for which code has already been provided), either with or without explicit SQL, but with the database code for each class centralized as class methods of that class.

A slightly more interesting variation of the *Classes As Storage Managers* strategy is to map tables to classes (as mentioned above), but use the automatic GLORP facilities to manage the database updates rather than explicitly coding class-side methods for it. But, this variation isn't really practical to show GLORP code for, because for GLORP to support this, you will have to code the mappings, descriptors, and descriptor systems to support the automatic facilities, and then learn how to use them within a session, and within a *UnitOfWork*. And once you've learned how to do that, then you are in a position to knowledgably use GLORP at a more sophisticated level than the *Classes As Storage Managers* level anyway, so there would be no point to the exercise.

But now you know a little more about what you need to cover to take GLORP to the next level—namely, sessions, descriptor systems, descriptors, mappings, and units of work.

GLORP Strategies – Level 3

Basic Mapping

Sessions[11]:

As mentioned before, the session normally decides whether writing a row for a given *DatabaseRow* should involve an *insert, an update*, or a *delete*, and it can generate a proper SQL string for any of those possibilities. Thus, the session has the responsibility (through DatabaseCommand objects) of generating that SQL. Sessions also manage the object caches, the descriptor systems (which will be explained later), and the units of work (also to be explained later). They also manage the descriptors and mappings indirectly (via the descriptor systems).

In short, GLORP is designed so that the bulk of all activity happens around a session. Thus, they are indespensable to the normal operation of GLORP.

Domain:

In order to map anything, you need to have domain object(s) that need to be mapped, and you need to have table(s) to map them to. We previously created a 'my_customer' table, so let's now create a *MyCustomer* class to map to it. For this example, your *MyCustomer* class needs to contain '*custNo*' and '*custName*'[12] instance variables.

GLORP can move data between the database and your domain objects, either using direct instance variable access, or by sending getters and setters. This is settable as an option in the DescriptorSystem (useDirectAccessForMapping). If you choose to use getters and setters, obviously those methods must exist, and must use standard Smalltalk naming conventions.

With this, we now have a specific class (*MyCustomer*) that we want to map to a specific table (*my_customer*). All that is needed now is to map it!

11 As of GLORP version 0.2.17, the *Session* class in GLORP has been renamed to *GlorpSession* due to name collision with *Session* in PWS on Squeak. Better namespace support would have rendered this a non-issue, but namespace support is currently only complete in VisualWorks (VisualAge only has partial namespace support, and Squeak only has experimental namespace code as of the time of this writing).

12 Notice that I've adopted standard Smalltalk conventions here instead of RDBMS conventions

Descriptor Systems:

Now that you have a *MyCustomer* class in the image, let's try asking a session to materialize one of them from the database:

```
GlorpSession new readOneOf: MyCustomer where: [:each | each custNo = 3].
```

I hope you didn't expect the above to succeed, because it will fail due to a "MessageNotUnderstood: descriptorFor:" error. Although you might not have realized exactly why it was going to fail, you should have expected some kind of failure because there is nothing there that tells the session where to read the *MyCustomer* instance from. But the interesting thing to notice here is what the debugger shows. So, look at the error in the debugger. You'll notice that #descriptorFor: is being sent to the *system* instance variable of the session, but *system* is currently nil. Then, if you browse senders on the `#system` message, you will discover that it is sent to the session from a wide variety of places. So evidentally, whatever the *system* is, and whatever it does, it is important!

So what should *system* be set to?

You can think of *system* as a sort of centralized controller for all of the mapping machinery. It should be an instance of a custom subclass of *DescriptorSystem*, and you need to explicitly set it.

To see this, try browsing class references to the *GlorpSession* class. I believe you will discover that every reference includes code that sends a *#system:* message to the new instance, and in every case I believe you will find that the argument to that message is an instance of a subclass of *DescriptorSystem*.

A properly set *system* is paramount to the functioning of GLORP. And to satisfy the *system* requirement, you must create a new subclass (of *DescriptorSystem*) of your own. Thus, the normal operation of GLORP does not merely depend upon creating instances of existing GLORP classes and telling them what to do, but you must also actually create your own new subclasses as well. That might not be exactly what you expected, but it is not difficult either.

Your *DescriptorSystem* subclass will need to implement the following methods:

#constructAllClasses - This returns a collection of your domain classes.
#allTableNames – This is an array of strings which are the names of the tables that you are mapping the domain classes to.

Now for each of the classes you will need to create a #descriptorFor{theName}: method. For example, suppose Company is one of the your classes —you will need to create a #descriptorForCompany: method. It will be passed in a descriptor instance, and must set its values to describe the mapping from the Company class to the database table(s). How to do this will be discussed in more detail later. Also, GLORP comes with an extensive test suite, which includes a number of different DescriptorSystems. You can look at these for examples of how to do many of the common things in GLORP.

Next, for each of the names in the #allTableNames array, you will need to create a #tableFor{theName}: method. For example, suppose your RDBMS has a 'CUSTOMER' table—you would create a

#tableForCUSTOMER: method that expects an instance of *DatabaseTable* for an argument, and will define fields for that table.

In addition to the information about tables and classes, the descriptor system holds onto some miscellaneous “global” information about the session. For example, this is where you tell GLORP whether to use direct instance variable access or accessor methods and what type of caching to use by default (individual descriptors can override this). It's also the place that remembers what platform (i.e. what database) we're using.

Descriptors:

For each persistent class, there is a descriptor. This is what tells GLORP how to read and write objects of this class. The descriptor is not actually a very complicated object, because most of the information is in the mappings described below. But the descriptor does keep hold of a few per-class pieces of information, notably

- tables - what tables does this class map to. It's common that a class maps to a single table, but not required. If you do specify a class with multiple tables, then you also need to specify how the rows from those tables are joined together. See the method addMultipleTableCriteria:, and the descriptor for GlorpPassenger in the tests.
- inherited reading - Inheritance can be a special case of objects mapping to multiple tables (although they can also be mapped within a single table). In that case you need some different APIs. See GlorpInheritanceDescriptorSystem for examples.

But mostly what you do with descriptors is add mappings to them...

Mappings:

A mapping defines the way in which one particular instance variable is mapped to the database. There are 3 different aspects to mapping. How we read it, how we write it, and what it means to query against it. There are several different kinds of mapping, and which one you use affects the way that these three things happen.

Direct Mappings:

The simplest mapping is *DirectMapping*. It specifies a direct correspondence between an instance variable and a field. The object containing the instance variable is not specified, because you later explicitly tell it what object to get/set values to or from. It will then move the data between that object and the table column as needed.

Here is an example of *DirectMapping* between the 'my_customer' database table (represented by the instance of *DatabaseTable* previously created, which I will just call *table* because that's what the previous code in the prior section above called it) and instances of the *MyCustomer* (also previously created above). Since that table has two columns, and since the *myCustomer* instance has two matching instance variables, we need two instances of *DirectMapping*:

```
    nameMapping := DirectMapping
```

```
                    from: #custName
                    to: (table fieldNamed: 'cust_name').
numberMapping := DirectMapping
                    from: #custNo
                    type: self platform integer
                    to: (table fieldNamed: 'cust_no').
```

By default, a direct mapping will make its best guess at the type of the instance variable based on the type of the corresponding database field. However, you can specifically tell it what type to use if you use the #from:type:to: method to create the instance, as in our second example. The argument to the type field is a Smalltalk class. The platform knows how to do common types of conversions. For example, if we said that the customer name was an instance of Symbol (not very realistic, but it's just an example), then GLORP would automatically handle the conversion back and forth between Symbols in memory and Strings in the database. If you need a conversion that isn't supported, then you'll have to extend the DatabasePlatform class to handle it.

Operations for a direct mapping are simple. To read, we convert from the database type to the Smalltalk type (we assume we've gotten the appropriate rows from somewhere). To write we do the reverse conversion and write it out. When we query, we convert the value and use it as part of the SQL statement. For example

```
    session readOneOf: Customer where: [:each | each custName = 'Foo'].
```

One to One Relationships:

Direct mappings only go so far. OO programs are about relationships between objects, and we need to be able to describe those. So, for example, suppose that we have an Order and a Customer. Each order knows the customer with which it's associated.

```
orderTable := self tableNamed: 'ORDER'.
customerTable := self tableNamed: 'CUSTOMER'
```

Note that “self” in this case is a descriptor system, and knows how to give us back tables. It's also important that we get back the identical instance every time we ask for a table.

```
mapping := OneToOneMapping new
             attributeName: #customer;
             referenceClass: Customer;
             mappingCriteria: (Join
                   from: (orderTable fieldNamed: 'CUSTOMER_ID')
                   to: (customerTable fieldNamed: 'ID')).
```

So, we've defined the attribute, exactly as we did for a direct mapping. We also need to know what sort of object will go in the instance variable, in this case a Customer. Then we have to define the database-level relationship that corresponds. In this case we define that to find an order's customer, we can join the field ORDER.CUSTOMER_ID with the field CUSTOMER.ID.

We're done. Based on just this information, GLORP knows how to read, write, and query against this relationship. Let's look at each individually.

To read, suppose that we have an Order already in memory. When we read it in GLORP will have populated all the instance variables of the object. If the instance variables are mapped with a direct mapping, then it will just put the values in right away. But if they're mapped with a relationship mapping, like the “customer” instance variable, then it will put a special Proxy in place (by default, there are ways to turn this off if you want to). The Proxy knows how to get the customer if it needs to. It knows, because when we read in the row for the order, we will have read in the CUSTOMER_ID field. That value is saved in the proxy, along with the query that says "get me the customer whose ID is X", where X is a parameter. When we access the proxy, it executes the query, providing the parameter that was saved with it, and reads the Customer.

Recall that when we write, we never write out individual objects, but rather all the objects that were affected by a particular "unit of work", a sort of object-level transaction. So chances are that we'll be writing out both the order and the customer at the same time. What the mapping criteria does is tell GLORP about a constraint on the values that are written out. So we know that for a particular order and its customer, the CUSTOMER_ID field and the ID field from the corresponding tables must be the same. When we combine that with the information about the ID field from its direct mapping in the descriptor for CUSTOMER, that's enough to know what values to write. If that doesn't make sense, don't worry too much about it, the important thing to know is that by specifying the information GLORP needs to read the right object from the relationship you've also specified enough information for it to write them out again.

Finally, the same information can be used for querying. Suppose we write a query that says

```
	session readManyOf: Order where: [:each | each customer zipCode = 12345]
```

This query is expressed at the object level, in terms of the order->customer relationship. But in the database this has to be translated into a join, something like

```
	SELECT ... FROM t1 ORDER, t2 CUSTOMER WHERE t1.CUSTOMER_ID=t2.ID AND
t2.zipCode=12345
```

Again, the mapping criteria we provided (note that it was described as a Join between two fields) is enough for GLORP to create the appropriate query automatically.

One To Many and Many To Many Relationships:

Similarly, we can describe relationships to multiple objects. Smalltalk doesn't distinguish at the implementation level whether a relationship is one to one or one to many. It's just collections. GLORP does distinguish these, based on the way they're implemented in the database.

Relational databases don't have a concept of collections. Instead, everything has to be represented as sets, joined together via foreign keys. So, in our previous example each Order had exactly one Customer. The reverse relationship, though, would be one to many. A Customer can have many Orders. In Smalltalk we represent this by a Customer object holding a collection (perhaps an OrderedCollection) of Orders, in addition to the Order having an instance variable "customer". In the relational database, though, we don't need any extra information. The foreign key from ORDER to CUSTOMER is enough to compute both relationships. In GLORP we could define this as

```
	aDescriptor addMapping: (
		OneToManyMapping new
			attributeName: #orders;
			referenceClass: Order;
			mappingCriteria: (Join
				from: (table fieldNamed: 'ID')
				to: ((self tableNamed: 'ORDER') fieldNamed: 'OWNER_ID'))).
```

where we assume that table is a temporary variable that's holding the CUSTOMER table. In this case we just specify the reverse join to what we did for the one to one relationship and tell GLORP that it's a one to many, and therefore represented as a collection. If we want to specify the kind of collection, we can do that as well by adding the line

```
        collectionType: Set;
```
This may be useful, because one of the properties of putting things into a relational database is that we lose ordering, because relational databases only deal with sets. If we want things to be ordered, then we need to specify the ordering as a sort, and it has to be based on some value from the data that's coming back, e.g.
```
        orderBy: [:each | each amount];
```
The ordering can be based on complex criteria, but you have to be a bit careful because if the ordering involves a join, you may end up missing values in the result unless you use an outer join.

orderBy: [:each | each billingAddress asOuterJoin zipCode].

This is one of those places where transparency can break down, and it can be important to understand a bit about what the database is actually doing, particularly the difference between Smalltalk's nil and the database's NULL, and the way joins work.

Often these kind of relationships are represented in the database with an additional layer of indirection: a *link table* in between the two main objects. This is required in many-to-many relationships, but can also be used to implement one-to-many's. In order to use this kind of relationship you use the GLORP ManyToManyMapping. so, suppose that we have classes Customer and BankAccount. A customer can have several bank accounts. An account can be joint, so it can have multiple account holders. We can represent this as
```
        ManyToManyMapping new
                attributeName: #accounts;
                referenceClass: BankAccount;
                mappingCriteria: (Join
                        from: (table fieldNamed: 'ID')
                        to: ((self tableNamed: 'CUSTOMER_ACCT_LINK')
                                fieldNamed: 'CUSTOMER_ID'))).
```
Note that this looks an awful lot like a one to many relationship, the only difference is that we use a different mapping class, and we refer to a link table rather than directly to the account table. There may or may not be a corresponding reverse relationship in the BankAccount class.

To do the join, a bit more information is needed, to know which field in the CUSTOMER_ACCT_LINK table matches the ACCOUNT table. However, for a simple link table, it's likely that GLORP can automatically figure out which field(s) to use. This is because GLORP knows about foreign key constraints in the database, and can examine them. In more complex cases, such as link tables with more than one foreign key to the target table, you may need to tell GLORP which fields to use, using
```
        mapping relevantLinkTableFields: (Array with: (linkTable fieldNamed:
'ACCOUNT_ID'))
```

Embedded Values:

We've already seen that GLORP can handle the case where a class maps to more than one table. But the opposite case is also important, where a single row can contain more than one instance. A common example for this sort of case is a Money object, which has instance variables for currency and amount. For efficiency, we don't want to store this in a separate Money table, but as two fields within the parent object. GLORP refers to this as an embedded value, where the word “embedded” here means that the object doesn't have its own identity (because it doesn't have its own primary key), but is associated with the identity of its parent object.

To make this more complicated, we often want to re-use these kind of embedded objects in multiple contexts. So, for example, an Order might have a money object associated with it, which is stored in fields in the ORDER table. However, a Purchase may also have a money object, stored in different fields. We'd like these to be instances of the same class in memory. And in fact, an Order might actually have multiple instances of money associated with it, perhaps one for the base amount and another for tax.

To support these kinds of cases, an embedded value mapping is allowed to define aliases for the fields it maps to. The basic descriptor for the Money object would refer to fields in one table (which need not even exist as a real table). When we define the embedded mapping we tell it the correspondence between those fields and the ones it should use in this table.

```
EmbeddedValueOneToOneMapping new
      attributeName: #amount
      referenceClass: Money
      fieldTranslation: (Join new
            addSource(transactionTable fieldNamed: 'MONEY_AMT')
            target: (moneyTable fieldNamed: 'AMOUNT');
            addSource: (transactionTable fieldNamed: 'MONEY_CURR')
```

```
                target: (moneyTable fieldNamed: 'CURRENCY'))
```
The presence of the OneToOne in the name suggests the possibility of embedded to-many relationships in a set of fields (ADDR1, ADDR2 etc.), but this is not yet supported.

Queries

We've seen some examples of queries, but haven't yet talked about how they work, or their syntax. Basically queries are specified using expression blocks. An expression block is a block which uses a restricted set of messages and operations, and can be translated into all or part of an SQL statement. For example

```
Query
    returningOneOf: Customer
    where: [:each | each id = 12].
Query
    returningManyOf: Customer
    where: [:each | each bankAccounts anySatisfy: [:eachAccount |
        eachAccount balance < 1000]].
```

This looks a lot like we were using a normal Smalltalk select statement, but in fact what is happening is that we execute the block once, build up a parse tree out of it, use the descriptors and mappings to figure out the corresponding database operations, and then generate and execute a query. The mechanism for parsing is to pass in a special doesNotUnderstand: proxy which simply records the messages to it, and then builds up a data structure out of it. This implies that certain operations won't work in these blocks. This includes operations that are optimized by Smalltalk, such as == (which doesn't make much sense in a database context anyway), and:, or: (use AND: and OR: instead, or the non-optimized & and |). We also can't do things like loops. Well, we can, but only certain kinds. Just recognize that none of the messages you're sending to the block argument or anything derived from it will actually do their operations. So

```
each bankAccounts do:
```

doesn't make sense, because we can't translate do: into database terms. The selector anySatisfy: is a special case that we can translate sensibly. There's a lot more we could say about what's possible, but the best bet for the moment is simply to look at how queries are used in the tests.

Queries also have a lot of options for indicating what to retrieve. The #retrieve: method allows you to bring back only particular fields from an object, or to automatically bring back related objects at the same time. #alsoFetch: is similar, but just puts the related objects into cache, not into the result set. #orderBy: allows you to specify the order in which the results will be brought back. #collectionType: brings back results in a particular kind of collection (by default an array).

To execute a query, you need the session

```
session execute: aQuery.
```

As a shortcut, the session has API's to directly create and execute a simple query

```
session readManyOf: Customer.
session readManyOf: Customer where: [:each | each id > 0].
```

Unit of Work

GLORP doesn't support any sort of API for writing a single object at a time. Instead, everything is done with a unit of work, a sort of object-level transaction. This is distinct from the database level transactions, although the two will often happen together. For example

```
session beginUnitOfWork.
session register: aCustomer.
aCustomer name: 'Foo'.
aCustomer addOrder: (SomeOrderGeneratingThing newOrder).
(Dialog confirm: 'Save changes?')
        ifTrue: [session commitUnitOfWork]
        ifFalse: [session rollbackUnitOfWork].
```

This is an extremely powerful paradigm. We only need to register the objects that we may be changing. Objects read in after the start of the unit of work are automatically registered. Objects that are related to things we register are also automatically registered. We can then make as many changes as we want. When we're finished, if we roll back the unit of work, then all the changes will go away and the objects will revert to the state they had when they were first registered. If we commit the unit of work, then those changes are automatically saved into the database.

Appendix

—

Class Reference

DatabaseAccessor

Inherits from: Object

Class Description

It is an instance of a platform-specific subclass of the *DatabaseAccessor* class that issues SQL code from your application to your RDBMS, whether that SQL is autogenerated (the preferred approach), or hand-coded (which is generally discouraged). However, while the DatabaseAccessor subclass is platform-specific, it is instantiated in a platform-neutral manner, through the DatabaseAccessor class itself. Thus, you don't need to know which subclass of DatabaseAccessor to instantiate.

This class has a few methods that are implemented as 'self subclassResponsibility'. And of course, the presence of such methods usually indicate that the class is an abstract superclass. Strictly speaking, this class *is* an abstract superclass, however from your perspective, you won't treat it as abstract, because you will typically send the #forLogin: instance creation message to this class, and it will select the concrete subclass to instantiate and hand back to you. Therefore, from your perspective, this class appears to violate the "*subclassResponsibility indicates abstract superclass*" rule. But it really isn't violating the rule.

To use this class, you first create an instance of Login (see the separate documentation for that class), and then create a platform-specific DatabaseAccessor subclass instance thus:

```
    accessor := DatabaseAccessor forLogin: aLogin.
```

Now, for Squeak, your 'accessor' from above will automatically end up being an instance of *SqueakDatabaseAccessor*. For VisualWorks, it will automatically be an instance of *VWDatabaseAccessor*. For Dolphin, it will automatically be an instance of *DolphinDatabaseAccessor*, and for VisualAge, it will automatically be an instance of *VA55DatabaseAccessor*.

For the most part, instance methods of this class are either DDL-oriented, or else overridden by subclasses (or both).

Instance Variables

connection	An instance of the database connection class that is used for your database. For Squeak, it will be an instance of the *PGConnection* class.
currentLogin	Contains an instance of *Login*, which is a data structure containing login parameters for your connection class.
platform	The current platform class. For Squeak this will be the class *PostgreSQLPlatform*.
logging	A *Boolean*. Indicates whether or not to log SQL statements to the transcript.

Instance Method Types

accessing	**#connection**
	#connectionClass
	#currentLogin
	#currentLogin:
	#platform
executing	**#createTable:ifError:**
	#doCommand:
	#doCommand:ifError:
	#dropConstraint:
	#dropTable:ifAbsent:
	#dropTableNamed:
	#dropTableNamed:ifAbsent:
	#dropTables
	#externalDatabaseAccessorSignal
initializing	**#initialize**
logging	**#log:**
	#logError:
login	**#login**
	#loginIfError:
	#showDialog:

Class Method Types

instance creation	**#classForThisPlatform**
	#forLogin:
	#new

Instance Methods

connection

A getter for the connection instance variable. There is no setter, and no other access to this instance variable exists in this class. Therefore, subclasses are expected to set it, and they typically will within their #loginIfError: method.

For Squeak, the *connection* will typically be an instance of PGConnection.

See also: #connectionClass

connectionClass

I don't see why you would ever care to call this method yourself. It is instead called from the `#loginIfError:` method, during database login, so that it knows what to set the *connection* instance variable to.

For Squeak, this method returns the PGConnection class, which is a class for representing the connection to a PostgreSQL database. For VisualWorks, it will return the VWDatabaseAccessor class. For Dolphin and VisualAge, it will return their respective database accessor classes.

In reality, this method just defers to `#connectionClassForLogin:`. However, curiously enough, that particular method only exists in subclasses. For Squeak, this *DatabaseAccessor* class will instantiate an instance of *SqueakDatabaseAccessor*, and for *SqueakDatabaseAccessor*, `#connectionClassForLogin:` returns returns the *PGConnection* class.

See also: #connection, #loginIfError:

createTable: *aGLORPDatabaseTable* **ifError:** *aBlock*

Attempts to create a database table based on the instance of *DatabaseTable* that is passed in. I personally don't see me using this method, except for within Sunit tests, as I prefer that my table creation code be scripts in external files.

currentLogin

Answers the *currentLogin* instance variable, which is meant to be an instance of Login, which is just a data structure describing the current database login parameters.

See also: #currentLogin:

currentLogin: *aLogin*

Sets the *currentLogin* instance variable. The parameter *aLogin* will be an instance of Login, which is just a data structure describing the current database login parameters.

See also: #currentLogin

doCommand: *aBlock*

Calls #doCommand:ifError:, but with 'self halt' expression as the error expression to execute if an error occurs.

See also: #doCommand:ifError:

doCommand: *aBlock* **ifError:** *errorBlock*

This is a simple method that just wraps the #executeSQLString: method inside of a standard #on:do: error handling block. *aBlock* is expected to either send the #executeSQLStatement: message to me, or else reference another method that sends it to me. Thus it is #executeSQLStatement: that actually does the work, and I expect *aBlock* to contain code that either directly or indirectly causes #executeSQLStatement: to be sent to me. All I do is wrap it (i.e., the #executeSQLStatement: call) within a standard error handler.

See also: #doCommand:, #executeSQLStatement:

dropConstraint: *aConstraint*

Drops a foreign key constraint. ForeignKeyConstraint is a class for specifying standard RDBMS foreign key constraints, and the argument is an instance of one.

Except for the SUnit tests, I doubt you will ever need this method.

dropTable: *aTable* **ifAbsent:** *aString*
dropTableNamed: *aString*
dropTableNamed: *aString* **ifAbsent:** *aBlock*
dropTables: *aCollectionOfTables*

Used for programmatically dropping tables from your RDBMS. It will generate the appropriate SQL for your RDBMS for dropping the table, and then cause that SQL to be sent to the RDBMS and committed. However, I personally prefer such code to be in external scripts.

externalDatabaseErrorSignal

Defined as 'self subclassResponsibility'.

For Squeak, though, the appropriate subclass will typically return the class *Error*. For VisualAge, it will typically return the class *ExError*. For VisualWorks, it returns an error signal class that is typically defined by some existing connection class within the existing VisualWorks EXDI database framework.

initialize

Returns self. This method exists because the #new method on the class side implements the simple '^super new initialize' pattern, and so we needed a default “do nothing” #initialize implementation.

log: *aString*

Sends *aString* to the Transcript on it's own line. At some point, this should probably be changed to write *aString* to a logStream, where that logStream can be directed anywhere (to the Transcript, to a file, internally to a ReadWriteStream, or whatever).

logError: *anErrorObject*

anErrorObject can be anything that responds to #printString. Of course, everything responds to #printString, because #printString is implemented in Object. But, typically *anErrorObject* will be an error exception instance.

login

Defers to #loginIfError:, with a standard Transcript message written as the error block.

loginIfError: *aBlock*

Implemented as 'self subclassResponsibility'. Subclasses will, of course, login to the database, and evaluate *aBlock* if an error occurs.

platforms

Returns the current platform class. For Squeak this will be the class PostgreSQLPlatform.

showDialog: *aString*

Implemented as 'self subclassResponsibility'. Subclasses will put up a modal dialog window containing *aString* as the dialog message. Modal dialogs are platform-specific, hence this behavior is deferred to the subclasses.

Class Methods

classForThisPlatform

Contains a big *case* statement for selecting and returning a platform-specific subclass of *DatabaseAccessor*. For Squeak, the class will be *SqueakDatabaseAccessor*.

new

Implemented as '^super new initialize'. Actually, you probably don't have a need to ever call this method, but instead just let #forLogin: call it.

forLogin: *aLogin*

aLogin is an instance of Login, which is just a data structure containing database login parameters.

This method calls #new, and then sets the *currentLogin* of that instance to *aLogin*.

DatabasePlatform

Inherits from: Object

Class Description

This is an abstract superclass.

Much of the support in DatabasePlatform (and all of it's subclasses) is DDL-oriented (Data Definition Langauge, which is the part of SQL that deals with database schema creation and manipulation). I believe that most of your DDL code is going to be for table creation, and I personally prefer to keep such code as external shell scripts, hence I personally don't see myself as using the DatabasePlatform class (or any of it's subclasses) at all, except to create a Login instance.

There appears to be some additional support in this class (as well as subclasses) to assist the GLORP machinery with generating proper SQL for the chosen RDBMS. For example, some RDBMS's might want column names to be upper case, and others lowercase. Some might prefer variable characters to be denoted as VARCHAR, while others might prefer VARCHAR2. If these differences exist, they can be handled transparantly via different subclasses of DatabasePlatform. The appropriate subclass will in turn be the concrete class that is used by the programmer when instantiating a Login instance.

In practice, however, GLORP limits it's SQL-generating machinery to (pretty much) simple and standard SQL. That means that subclasses of this class do very little.

In any case, you can pretty much ignore this class while learning the GLORP framework.

DatabaseTable

Inherits from: Object

Class Description

I am used to describe a table within the RDBMS to the GLORP framework. The table description will include things like: what the column names and column data types are, what column or columns comprise the table key, what the foreign keys are and what other tables and columns they map to.

This description is necessary in order for GLORP to generate the appropriate SQL code for the table. In other words, it is necessary if you want me to create the table for you, or if you want me to drop the table for you, or if you want me to tell the database about the primary and foreign keys that I know of, etc.

So, as a storage place for the table definition, I am mainly like a data structure, with no real behavior. However, because I can also generate table creation SQL based on that table definition, I am more than just a data structure. I'm a real object.

I have some other "real behavior", too. For example, I can tell a DatabaseAccessor that I want the ForeignKeyConstraints that I know about to be dropped from the database. If I do that, then obviously somewhere along the line some other object will end up generating and issuing an appropriate 'ALTER TABLE' SQL statement to the database to accomplish it. So my actions can cause other objects to generate SQL of their own, as well.

Instance Variables

name	The name of the database table that I represent, as a string.
creator	The schema in which the table exists. This has been renamed to *schema* in current versions of GLORP. If it is set, then the table will print a schema.tableName rather than just tableName.
fields	A collection of instances of *DatabaseField* that represent the columns of the table that I represent.
primaryKeyFields	A collection of instances of *DatabaseField* that represent the primary key columns of the table that I represent.
foreignKeyConstraints	A collection of instances of *ForeignKeyConstraint*, which represents the foreign key relationships in the table that I represent.
parent	Appears to be intended for an instance of *DatabaseTable*, but I don't think it is currently used. See the discusion about the `#creator:` method for more information.

Instance Method Types

accessing	**#creator**
	#creator:
	#fields
	#foreignKeyConstraints
	#name
	#name:
	#parent
	#parent:
	#primaryKeyFields
	#qualifiedName
	#sqlTableName
create/delete in db	**#creationStringFor:**
	#dropForeignKeyConstraintsFromAccessor:
	#dropFromAccessor:
	#primaryKeyConstraintName
	#primaryKeyUniqueConstraintName
	#printDelimiterOn:
	#printFieldsOn:for:
	#printForeignKeyConstraintsOn:for:
	#printPrimaryKeyConstraintsOn:for:
fields	**#addField:**
	#addForeignKeyFrom:to:
	#createFieldNamed:type:
	#fieldNamed:
	#newFieldNamed:
initializing	**#initialize**
printing	**#printOn:**
	#printSQLOn:withParameters:
	#sqlString
testing	**#hasCompositePrimaryKey**
	#hasConstraints
	#hasFieldNamed:
	#hasForeignKeyConstraints
	#hasPrimaryKeyConstraints
private/fields	**#addAsPrimaryKeyField:**

Class Method Types

instance creation	**#named:**
	#new

Instance Methods

addAsPrimaryKeyField: *aDatabaseField*

A private message sent from instances of DatabaseField that you have sent #bePrimaryKey to. So, you tell the field (*aDatabaseField*) to be a primary key, and it takes care of informing me, the table (*aDatabaseTable*). But once this has been done, there is no way to have it undone. This means that if you need to change the primary key, you need to start over and create a new *DatabaseTable* instance to represent the changed table. That's actually not as restrictive as it sounds, when you realize that many (most? all?) RDBMS's don't let you change the primary key of a table anyway. For such RDBMS's, if you really need to change the primary key, you first must unload the data from the table that you want to change, then destroy the old table, then recreate a new table with the desired primary key change, and then reload the data into the new table. So this really is similar to the need to destroy your current *DatabaseTable* instance, and then recreate a new one with the changed primary key. So it's not really any more onerous than what RDBMS's make you do anyway.

The table (me) can have a composite of several fields that together comprise the primary key (in other words, the *primaryKeyField* instance variable is a collection).

See also: #hasCompositePrimaryKey

addField: *aDatabaseField*

Adds *aDatabaseField* to the collection of fields recorded for this table. Returns aDatabaseField.

The instance variable *fields* is an OrderedCollection. That means that the same database field could be added multiple times via this method, but that shouldn't ever happen because this method should probably be considered private (and marked that way). Use the method `#createFieldNamed:type:` to add a field—it will call this method as needed.

See also: #createFieldNamed:type:

addForeignKeyFrom: *sourceField* **to:** *targetField*

Creates a ForeignKeyConstraint instance from the arguments, and adds that instance to the ForeignKeyConstraint collection.

You can add the same ForeignKeyConstraint multiple times, so be careful.

createFieldNamed: *aString* **type:** *dbType*

The *dbType* argument is is one of the types returned from the class methods such as `#int4, #sequence, #varChar`, for the subclasses of *DatabasePlatform*. Hmmm, actually, those three seem to be the *only* supported types!

creationStringFor: *aDatabaseAccessor*

Generates and returns a SQL "CREATE TABLE" statement (as a string) based on the receiver's known definition of that table. The generated SQL statement will be appropriate for your chosen RDBMS, which is why the *aDatabaseAccessor* argument must be passed in.

creator

A private getter for the *creator* instance variable, which is initialized to an empty string '' in the #initialize method. This is the user space name, as described in the setter message #creator:.

See also: #creator:

creator: *aString*

A private setter for the *creator* instance variable. (This method has been renamed to *schema* in current versions of GLORP, to be more consistent with standard database terminology)

Some databases (such as Oracle) put tables within a user space. For such databases, a table reference outside of the current user space must be prefixed with the user space name (and tables within the current user space can optionally be prefixed with the user space name). This user space name is sometimes referred to as the *creator* in other literature. For example, *purchasing.purchase_orders* references the *purchase_orders* table within the *purchasing* user space, so in this case the *creator* will be set to *purchasing*.

If the creator is set, and if the currently active *DatabasePlatform* subclass is for a database that requires such prefixing, then any generated SQL will have the table names prefixed by the creator, followed by a period, as the above example shows.

For a given RDBMS, creators can often be nested. For example, the reference *retail.purchasing.purchase_orders* references the *purchase_orders* table from within the *purchasing* user space that in turn is within the *retail* user space. For such a nested reference, the `#sqlTableName` method will return the entire reference. For such situations, the *parent* instance variable and associated methods (getter and setter) appear to be the planned design to handle it, thus allowing a hierarchy of *DatabaseTable* instances. However, I don't think that this is completely implemented yet, as I don't think the *parent* attribute is implemented yet.

For PostgreSQL, the *creator* does not appear to be used (or needed). Thus, I really don't have any experience with it, which means the above discussion could contain errors.

See also: #name:, #sqlTableName

dropForeignKeyConstraintsFromAccessor: *aDatabaseAccessor*

I will iterate through all of ForeignKeyConstraints that I know about, and tell each of them to drop themselves from the database. Consequently, those ForeignKeyConstraints will either generate SQL to do that, or somehow cause SQL to be generated, and then sent to the RDBMS indicated by the argument *aDatabaseAccessor*.

See also: #foreignKeyConstraints

dropFromAccessor: *aDatabaseAccessor*

When you send this message to me, I will tell the RDBMS to drop the table I represent. I generate SQL to do that, and send it to the RDBMS indicated by the argument *aDatabaseAccessor*. If necessary, I will first generate SQL (and send it to the RDBMS) for dropping primary key constraints before asking the RDBMS to drop the table.

fieldNamed: *aString*

I will search the *OrderedCollection* of instances of *DatabaseField*, looking for one that has a name matching *aString*. If none are found, I throw an *Error* exception, with the string 'Object is not in the collection'. If you want to test if the field exists before calling me, use `#hasFieldNamed:` to test.

See also: #createFieldNamed:type:, #hasFieldNamed:

fields

Returns the *OrderedCollection* of instances of *DatabaseField*, one field for each column of the table represented by me.

See the discussion about the message #foreignKeyConstraints to see why I think this method should either implement what I call “collection protection”, or else be eliminated entirely.

See also: #fieldNamed:, #createFieldNamed:type:, #hasFieldNamed:

foreignKeyConstraints

A private getter for this instance variable.

For Squeak, the *foreignKeyConstrants* will be an OrderedCollection of instances of ForeignKeyConstraints.

Note: if this method truly is supposed to be private, as it indeed appears to be, my own preference would be to eliminate this method entirely, and let senders of this message reference *foreignKeyConstraints* directly. In other words, I don't believe in private accessors. Furthermore, my philosophy is that public accessors that return collections should employ “collection protection”, which is a pattern whereby you always return a copy of the collection instead of the original. That way, other objects cannot mess around with the contents of a collection that this object intends to manage—i.e., the collections are “protected” from external manipulation.

See also: #addForeignKeyFrom:to:, #dropForeignKeyConstraintsFromAccessor:

hasCompositePrimaryKey

Tests to see if the *primaryKeyFields* collection has a size greater than 1, which if true indicates that there are more than one field that has been designated as a primary key.

See also: #primaryKeyFields

hasConstraints

Tests to see if there are any primary or foreign key constraints defined.

See also: #hasForeignKeyConstraints, #hasPrimaryKeyConstraints

hasForeignKeyConstraints

Tests to see if there are any foreign key constraints defined.

See also: #hasConstraints, #hasPrimaryKeyConstraints

hasPrimaryKeyConstraints

Tests to see if there are any primary key constraints defined.

See also: #hasForeignKeyConstraints, #hasConstraints

hasFieldNamed: *aString*

Tests to see if I know about any field in the table I represent whose field name is *aString*. You might want to test if the field exists before calling `#fieldNamed:` to get the field.

See also: #createFieldNamed:type:, #hasFieldNamed:, #fieldNamed:

initialize

This method is automatically called when a new instance is created. It just sets the various collection instance variables to empty collections, and initializes the *creator* instance variable to an empty string.

name

A private getter for the *name* instance variable. This is the base table name.

See also: #name:

name: *aString*

A private setter for the *name* instance variable. This is the base name of the table. For the *purchasing.purchase_orders* example previously given, the table name is *purchase_orders*.

See also: #creator:, #sqlTableName

newFieldNamed: *aString*

Just generates an error exception telling you to instead use `#createFieldNamed:type:`. I don't understand why we don't just delete this method. It's existence isn't for polymorphic reasons, as this message name is only implemented in this class, and this method doesn't do anything.

See also: #creatFieldNamed:type:

parent

A getter for the *parent* instance variable. I don't think this is actually being used yet. See the discussion about #creator: for more information.

See also: #creator:

parent: *aDatabaseTable*

A setter for the *parent* instance variable. I don't think this is actually being used yet. See the discussion about #creator: for more information.

See also: #creator:

primaryKeyConstraintName

I return '^ self name, '_PK', which is my table name following by '_PK'.

The constraint name is later used by other mechanisms to generate appropriate SQL that defines and/or drops constraints in the database.

See also: #primaryKeyUniqueConstraintName

primaryKeyFields

A getter for the *primaryKeyFields* instance variable, which is an Array. The instances within the array are instances of *DatabaseField*, and they describe a field, or column, within the table that I represent .

See my discussion on “collection protection” for the *foreignKeyContraints* method for more comments.

See also: #foreignKeyConstraints

primaryKeyUniqueConstraintName

I return '^ self name, '_UNIQ', which is my table name following by '_UNIQ'.

The unique constraint name is later used by other mechanisms to generate appropriate SQL that defines and/or drops constraints in the database. It is used for SQL that requires the UNIQUE keyword for the CONSTRAINT in the SQL.

See also: #primaryKeyConstraintName

printDelimiterOn: *aStream*

You can consider this method to be private, as I see no reason you will need to send this message yourself.

The delimiter I print is the standard SQL delimiter that is used to separate field names within the generated SQL. Currently this is always the comma character, and I don't particular see a situation when it would ever need to be different from that. So, I'm not quite sure why this method even exists.

printFieldsOn: *aCreationStream* **for:** *aDatabaseAccessor*
printForeignKeyConstraintsOn: *aCreationStream* **for:** *aDatabaseAccessor*

These two methods probably should be considered private methods used by the `#creationStringFor:` method. While that method generates the SQL for a 'CREATE TABLE' statement, these methods generate the SQL for the fields portion of that SQL statement, and the foreign key constraints portion of that SQL statement respectively. The SQL is written to the stream specified in *aCreationStream*. Since the SQL might be database-specific, *aDatabaseAccessor* must also be passed in.

printOn: *aStream*

Standard *#printOn:* statement. Puts the table name enclosed in parenthesis into the stream, or an empty string if the table name is not yet defined. Standard stuff.

printPrimaryKeyConstraintsOn: *aCreationStream* for: *aDatabaseAccessor*

Just like `#printForeignKeyConstraintsOn:for:`, only for the primary key instead of foreign key.

See also: #printForeignKeyConstraintsOn:for:

printSQLOn: *aStream* **withParameters:** *aDictionary*

This appears to me to be an unfinished method, and I'm not sure what it's design intent is. Right now it just prints the table name onto *aStream*. The *aDictionary* argument is ignored.

qualifiedName
sqlString

Both of these methods just return the *name,* which is the table name. I'm not sure why they exist as a separate methods, unless it is to preserve polymorphic behavior with other objects. Both *DatabaseTable* and *DatabaseField* implement the instance method `#qualifiedName`, but only *DatabaseTable* implements `#sqlString`. So, either there is some unfinished business here, or else these are potentially some superfulous methods.

See also: #name

sqlTableName

Returns the complete name reference of the table, complete with user name (i.e., *creator*, if appropriate for your RDBMS), and nested table path (whenever they become implemented).

In other words, this is the complete table reference as appropriate in any list of tables in any given SQL statement for your particular RDBMS.

See also: #creator:

Class Methods

named: *aString*

Creates a new instance, and sets the *name* (i.e., the table name) to *aString*

new

Implemented as '^super new initialize'. Actually, you probably don't have a need to ever call this method, but instead just let #forLogin: call it.

DescriptorSystem

Inherits from: Object

Class Description

I am used to

Instance Variables

descriptors	A collection of
typeResolvers	A collection of
tables	A collection of instances of *DatabaseTable*.
session	An instance of *GlorpSession*.
cachePolicy	Normally an instance of *CachePolicy*, but can be changed to some other cache policy (which I assume would always be a subclass of *CachePolicy*).
platform	An instance of one of the subclasses of *DatabasePlatform*. For Squeak, this will be *PostgreSQLPlatform.*

Instance Method Types

accessing	**#allClasses**
	#allDescriptors
	#allTables
	#cachePolicy
	#cachePolicy:
	#defaultTracing
	#platform
	#session
	#session:
api	**#descriptorFor:**
	#existingTableNamed:
	#hasDescriptorFor:
	#tableNamed:
	#typeResolverFor:
private	**#initialize**
	#newDescriptorFor:
	#newTableNamed:
	#newTypeResolverFor:

Class Method Types

instance creation	**#forPlatform:**
	#new

Instance Methods

allClasses

Sends 'self allClassNames', and then looks up the classes associated with those names, and returns a collection of those classes.

#allClassNames is only implemented in subclasses. There should probably be a #subclassResponsibility implementation of it in this class.

In the subclass, #allClassNames needs to return a collection of names (as symbols) of the domain classes that we are using GLORP to map to the RDBMS.

See also: #allTables

allDescriptors

Returns a collection of descriptors (instances of *Descriptor*) that each describe a mapping of a class (one of those returned via `#allClasses`) to the tables (returned via `#allTables`). A descriptor in turn will know what class it maps, and all the details of how it maps it.

See also: #allClasses, #allTables

allTables

Sends 'self allTableNames', and then looks up the table associated with those names, and returns a collection of those tables. The “tables” are instances of *DatabaeTable*, and the lookup for those instances occurs by #tableNamed: to self. #tableNamed: in turn looks at the internal *tables* collection for tables with that name.

#allTableNames is only implemented in subclasses. There should probably be a #subclassResponsibility implementation of it in this class.

In the subclass, #allTableNames needs to return a collection of names (as symbols) of the tables that exist in the RDBMS that we are using GLORP to map to the domain objects.

See also: #allClasses

cachePolicy

Answers the *cachePolicy*. The *cachePolicy* is the default cache policy that will be used for descriptors that don't specify their own policy. The default is can be set via `#cachePolicy:`, but if it is not set, it is found by sending *CachePolicy>>default,* and then set based on what that message returns.

Hmm, this means that if the default *cachePolicy* isn't set, then the default *cachePolicy* is set to the default cache policy of the *CachePolicy*. Interesting.

defaultTracing

Answers a new instance of the *Tracing* class.

descriptorFor: *aClassOrObject*

Answers the descriptor for the argument. If need be, a new descriptor is created (the descriptors are kept in the *descriptors* collection internally).

existingTableNamed: *aString*

Searches the tables collection for a DatabaseTable instance whose name is aString. Throws an error exception by sending #error: if one is not found (why not just let the normal 'key not found' error exception happen instead of explicitly sending #error: in this case? Also, shouldn't we implement #existingTableNamed:ifAbsent:?)

hasDescriptorFor: *aClassOrObject*

This method seems to be broken, because it will always return *true*. The reason it returns *true* is it uses `#descriptorFor:,` which will create the descriptor if it needs to. Consequently the descriptor will always exist once you call this method.

initialize

Standard initialize stuff.

newDescriptorFor: *aClass*

Private. You should never call this yourself. Use `#descriptorFor:` instead.

Answers a new instance of the *Descriptor,* initializing it as a descriptor for *aClass*. Subclasses must create a #descriptorFor*aClass* method for this to work (where *aClass* is the actual class name).

See also: #descriptorFor:

newTableNamed: *aString*

Private. You should never call this yourself. Use #tableNamed: instead.

Answers a new instance of the *DatabaseTable,* initializing it as needed for that table in the RDBMS. Subclasses must create a #tableFor*aString:* method for this to work (where *aString* is the actual table name), and that method is what is used to initialize the *DatabaseTable* instance that is created here.

NOTE: #tableFor*aString*: expects you to pass to it the uninitialized *DatabaseTable* instance—well, uninitialized all except for the name. Thus, the table name information is duplicated, because it exists in the argument, plus it exists as a part of the #tableFor*aString*: method name. So, why not just let #tableFor*aString*: create the *DatabaseTable* instance itself, and thus eliminate the need to pass in an argument?

See also: #tableNamed:

newTypeResolverFor: *aClass*

Private. You should never call this yourself. Use #typeResolverFor: instead.

Answers a type resolver for *aClass*. Subclasses must create a #typeResolverFor*aClass* method for this to work (where *aClass* is the actual class name)

See also: #typeResolverFor:

platform

Raw getter for the *platform* instance variable.

platform: *dbPlatform*

Raw setter for the *platform* instance variable. *dbPlatform* will be an instance of one of the subclasses of *DatabasePlatform*.

session

Raw getter for the *session* instance variable.

session: *anObject*

Raw setter for the *session* instance variable. *session* will be an instance of *DatabaseSession*.

tableNamed: *aString*

Searches the *tables* collection for an instance of *DatabaseTable* whose name is *aString*. If one is not found, one is created and put into the *tables* collection. Returns that instance.

Actually, a new instance of *DatabaseTable* is only created if the subclass properly implemented a #tableFor*aString*: method. Otherwise it looks like a *MessageNotUnderstood* error is generated.

See also: #newTableNamed:

typeResolverFor: *aClassOrObject*

Searches the *typeResolvers* collection for a type resolver for the argument. If one is not found, one is created and put into the *typeResolvers* collection. Returns that instance.

Actually, a type resolver is only created if the subclass properly implemented a #typeResolverFor*aClass* method, where *aClass* is the name of the class of the argument. Otherwise it looks like a *MessageNotUnderstood* error is generated.

See also: #newTypeResolverFor:

Class Methods

forPlatform: dbPlatform

Implemented as '^super new initialize; platform: dbPlatform'.

new

Implemented as '^super new initialize'.

GlorpSession

Inherits from: Object

Class Description

The *GlorpSession* is the heart of the applications interface to the GLORP layer. It manages the database accessor, the *UnitOfWork* transaction management, the object/table/row caches, and the object-to-relational mapping model, where the mapping model includes the descriptors, descriptor systems, and actual mappings. In short, it manages just about everything in GLORP, where "manage" here means that anywhere your application accesses those "managed" resources, you almost always do it via a *GlorpSession*. In some cases, the *GlorpSession* manages the resource directly, and in other cases, you ask the *GlorpSession* for the resource, and then you communicate with that resource directly. But in pretty much all the cases, your doorway into the rest of GLORP is via the *GlorpSession*.

Instance Variables

system	An instance of a custom subclass of *DescriptorSystem* that represents your complete O/R mapping model.
currentUnitOfWork	An instance of *UnitOfWork* if we are currently within a unit of work. Otherwise nil (because #commitUnitOfWork sets it to nil).
cache	A *CacheManager*, which is a handle on the cache subsystem. Private.
accessor	An instance of a custom subclass of *DatabaseAccessor*. For Squeak, this will be an instance of *SqueakDatabaseAccessor*..
applicationData	I don't think this is currently used.

Instance Method Types

accessing	**#accessor**
	#accessor:
	#applicationData
	#applicationData:
	#system
api	**#descriptorFor:**
	#hasDescriptorFor:
	#register:
	#registerAsNew:
	#system:
api/queries	**#delete:**
	#execute:
	#hasExpired:

	#readManyOf:where:
	#readOneOf:where:
	#refresh:
api/transactions	**#beginTransaction**
	#beginUnitOfWork
	#commitTransaction
	#commitUnitOfWork
	#hasUnitOfWork
	#rollbackTransaction
	#rollbackUnitOfWork
caching	**#cacheAt:forClass:ifNone:**
	#cacheAt:put:
	#cacheContainsObjectForClass:key:
	#cacheContainsObjectForRow:
	#cacheLookupForClass:key:
	#cacheLookupObjectForRow:
	#cacheRemoveObject:
	#hasExpired:key:
	#hasObjectExpiredOfClass:withKey:
	#isRegistered:
	#lookupRootClassFor:
copying	**#copy**
	#postCopy
events	**#sendPostFetchEventTo:**
	#sendPostWriteEventTo:
	#sendPreWriteEventTo:
initializing	**#initialize**
	#initializeCache
	#reset
internal/writing	**#createDeleteRowsFor:in:**
	#createRowsFor:in:
	#shouldInsert:
	#sqlDeleteStringFor:
	#sqlInsertStringFor:
	#sqlStringFor:
	#sqlUpdateStringFor:
	#tablesInCommitOrder
read/write	**#filterDeletionFrom:**
	#filterDeletionsFrom:
	#writeRow:
testing	**#isNew:**
	#isUninstantiatedProxy:
private	**#expiredInstanceOf:key:**
	#privateGetCache
	#privateGetCurrentUnitOfWork
	#realObjectFor:

Class Method Types

instance creation	**#forSystem:**
	#new

Instance Methods

accessor

Answers the database accessor.

See also: #accessor:

accessor: *aDatabaseAccessor*

Sets the database accessor, which is a custom subclass of *DatabaseAccessor*. For Squeak, this will be an instance of *SqueakDatabaseAccessor*.

applicationData

I can't tell that this is being used for anything.

applicationData: *anObject*

I can't tell that this is being used for anything.

beginTransaction

Tells the database accessor to begin a database transaction. You normally wouldn't send this yourself, instead you would normally send #beginUnitOfWork. However, suppose you want GLORP to write and/or update rows in the database, including an explicit #commitUnitOfWork, but with the intent of rolling back the transaction so that they are not permanent after committing the unit of work. In that case, you would first explicitly send #beginTransaction before sending #beginUnitOfWork. If the *UnitOfWork* notices you are already in a transaction, it won't commit it's changes. Otherwise it will start a transaction when you #beginUnitOfWork, and commit it with #commitUnitOfWork (or roll it back with #rollbackUnitOfWork).

But usually you don't send #beginTransaction yourself.

See also: #beginUnitOfWork

beginUnitOfWork

Creates a *UnitOfWork* instance and sets the '*currentUnitOfWork*' to it. See the Object Reference for the *UnitOfWork* class for more information.

cacheAt: *aKey* **forClass:** *aClass* **ifNone:** *failureBlock*

Probably should be considered a private method.

cacheAt: *keyObject* **put:** *valueObject*

Probably should be considered a private method.

cacheContainsObjectForClass: *aClass* **key:** *aKey*

Probably should be considered a private method.

cacheContainsObjectForRow: *aDatabaseRow*

Probably should be considered a private method.

cacheLookupForClass: *aClass*

Probably should be considered a private method.

cacheLookupObjectForRow: *aDatabaseRow*

Probably should be considered a private method.

cacheRemoveObject: *anObject*

Probably should be considered a private method.

commitTransaction

Tells the database accessor to commit the transaction. You probably won't send this directly, just as you probably won't send #beginTransaction directly.

See also: #beginTransaction

commitUnitOfWork

See the documentation on the *UnitOfWork* class for more information on this. When done, though, the '*currentUnitOfWork*' instance variable will be nil.

See also: #beginUnitOfWork

copy

Makes a shallow copy, and then runs #postCopy afterwards (which in turn initializes the cache and resets the unit of work). Used to clone the session, although I'm not sure why you would want to do that.

createDeleteRowsFor: *anObject* **in:** *rowMap*

Private. Sent from the *UnitOfWork*.

createRowsFor: *anObject* **in:** *rowMap*

Private. Sent from the *UnitOfWork*.

delete: *anObject*

anObject is a domain object that GLORP is mapping to the RDBMS. This method marks *anObject* for deletion from the RDBMS, and it does it within a *UnitOfWork*. It begins a *UnitOfWork* if one has not already been started. When the *UnitOfWork* is committed, the rows in the RDBMS that *anObject* is mapped to will be deleted, and *anObject* is also removed from the object cache. The specific rows that get deleted from the RDBMS is entirely dependent upon the mappings.

descriptorFor: *anObject*

Returns the descriptor for *anObject*. The descriptor in turn manages the specific mappings that map *anObject* to specific rows within the RDBMS. Therefore, via the descriptor, you can fetch the mappings if you desire.

You need to know what descriptors are (and how to create them) in order to do the O/R mapping correctly. But once you've set up your descriptors, and handed them to your *system* (a *DescriptorSystem* subclass), which in turn is handed to the *GlorpSession* when you create it, you normally don't need to deal with descriptors afterwards. Hence, I doubt you'll use this method, excepts possibly in test cases that test internals.

execute: *aQuery*

aQuery is a concrete subclass of *Query*. *Query* provides a programmatic way to put together queries for domain objects that happen to reside in the RDBMS. This method will execute the query once it is built, and returns the result of executing it.

You might be tempted to think of this method in terms of a SQL query which returns rows from the database. However, this method is only loosely related to that idea. This method does not return rows from a database. It returns one or more domain objects which happen to prove true for the query. Hence, this method is more like the standard Smalltalk #select: method for collections. As you know, for #select: a collection of objects is returned for which the argument block evaluates true. In a similar vein, this method returns a collection of objects (or possibly a single object) for which the argument query evaluates true.

Of course, since the objects returned by this method happen to be persistent domain objects, GLORP will fetch rows from the database on an as-needed basis in order to build the domain objects needed to satisfy the request (and it will do that based on the mappings). Hence, GLORP might fetch rows from the database in response to this method (or it might just get domain objects from the object cache instead), but the semantics of this method are that it returns domain objects—not rows.

expiredInstanceOf: *aClass* **key:** *keyObject*

Private cache method.

filterDeletionFrom: *anObject*

Should be considered private. Potentially *anObject* has been marked for deletion by the *UnitOfWork*. This method is used by the #execute: method to filter out such objects that have been marked for deletion.

filterDeletionsFrom: *aCollection*

Should be considered private.

See also: #filterDeletionFrom:

hasDescriptorFor: *aClass*

Does the descriptor exist? Yes or no (true or false).

See also: #descriptorFor:

hasExpired: *anObject*

Should be considered private. The cache subsystem uses this (and some unit tests do as well).

hasExpired: *aClass* **key:** *key*

Should be considered private. The cache subsystem uses this (and some unit tests do as well).

hasObjectExpiredOfClass: *aClass* **withKey:** *key*

Should be considered private. The cache subsystem uses this (and some unit tests do as well).

hasUnitOfWork

Is the 'currentUnitOfWork' not nil?

See also: #beginUnitOfWork

initialize

Initializes the cache.

See also: #initializeCache

initializeCache

Creates a *CacheManager*.

isNew: *anObject*

If the cache does not contain an object for the class of *anObject*, then *anObject* is new. I don't see why you would ever need to send this yourself, as it looks like it is private to the object registration subsystem.

isRegistered: *anObject*

GLORP ignores any object that is not reachable from a registered object. So, is *anObject* registered?

isUninstantiatedProxy: *anObject*

Private. Only sent from #register:

lookupRootClassFor: *aClass*

Private. Sent from the *CacheManager*.

postCopy

Private. Sent from #copy.

See also: #copy

privateGetCache
privateGetCurrentUnitOfWork

Private.

readManyOf: *aClass* **where:** *aBlock*

Very similar to #execute:, but builds a *ReadQuery* for you from the arguments. In fact, uses #execute: to execute the query once it is built.

See also: #execute:

readOneOf: *aClass* **where:** *aBlock*

Like #readManyOf:where:, but stops after the first object is found.

See also: #readManyOf:where:

realObjectFor: *anObject*

Private. If *anObject* is a proxy, the object it represents is materialized and replaces the proxy. Returns the materialized real object, or *anObject* if it is already a materialized real object.

refresh: *anObject*

If *anObject* is in the object cache, it's instance variables are refreshed from the database. If *anObject* is not already in the object cache, it is materialized into the cache from the database.

register: *anObject*

Registers *anObject* with the current UnitOfWork, so that GLORP can manage it's persistence in the database.

See also: #registerAsNew:

registerAsNew: *anObject*

Does stuff.

See also: #register:

reset

Does stuff.

See also: #

rollbackTransaction

Does stuff.

See also: #

rollbackUnitOfWork

Does stuff.

See also: #

sendPostFetchEventTo: *anObject*

Does stuff.

See also: #

sendPostWriteEventTo: *anObject*

Does stuff.

See also: #

sendPreWriteEventTo: *anObject*

Does stuff.

See also: #

shouldInsert: *aDatabaseRow*

Does stuff.

See also: #

sqlDeleteStringFor: *aDatabaseRow*

Does stuff.

See also: #

sqlInsertStringFor: *aDatabaseRow*

Does stuff.

See also: #

sqlStringFor: *aDatabaseRow*

Does stuff.

See also: #

sqlUpdateStringFor: *aDatabaseRow*

Does stuff.

See also: #

system

Does stuff.

See also: #

system: *aSystem*

Does stuff.

See also: #

tablesInCommitOrder

Does stuff.

See also: #

writeRow: *aDatabaseRow*

Does stuff.

See also: #

Class Methods

#forSystem:

Does stuff.

#new

Implemented as '^super new initialize'.

Login

Inherits from: Object

Class Description

This class is not particularly important for understanding the GLORP framework. It is just a data structure with no behavior. It contains the individual data elements that are needed for connecting to the database of your choice. The data elements are defined as:

1. database
2. username
3. password
4. connectString

There are simple getter and setter methods for each of these four attributes. That's all this class has or does.

As an example, for the PostgreSQL database, you might create an instance of Login thus:

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

The above example would be for accessing the 'db' PostgreSQL database residing on the machine named 'host', and with user name of 'username' and password of 'password'. Obviously your installation would have different names for each of these four items.

See the *DatabaseAccessor* class documentatin for more informatin.

ObjectTransaction

Inherits from: Object

Class Description

This can be considered a private class for the exclusive use of the *UnifOfWork* class. It implements most of the undo mechanism, and is also the actual registrar for registering objects to GLORP.

This means that the *UnitOfWork* class must be the primary handler of database transactions. If you instead bypass the *UnitOfWork* functionality, thinking to instead go straight to lower layers to explicitly issue transaction begins, commits or aborts (perhaps because you think all you need is a little bit of SQL executed, and not the entire GLORP framework), you will not be able to use the undo mechanism built into GLORP. Plus, since objects won't be registered to GLORP, you will not be able to use any of the automatic mechanisms of GLORP.

The lower layers would include any other mechanism that allows you to issue SQL straight to the datbase. This would include the EXDI layer (for VisualWorks), or straight to the `PGConnection>>execute:` code for PostgreSQL on Squeak, or even to other GLORP layers like `DatabaseAccessor>>doCommand:`.

Instance Variables

undoMap	An identity dictionary. Each element is an original object, and it's copy. Should the original object change, and you wish to restore it, you can restore it from the copy

Instance Method Types

accessing	**#undoMap**
begin/commit/abort	**#abort**
	#begin
	#commit
registering	**#isRegistered:**
	#realObjectFor:
	#register:
	#registerTransientInternsalsOfCollection:
	#registeredObjectsDo:
	#requiresRegistrationFor:
private/registering	**#instanceVariablesOf:do:**
	#shallowCopyOf:ifNotNeeded:

private/restoring	**#isShapeOf:differentThanThatOf:**
	#restoreIndexedInstanceVariablesOf:toThoseOf:
	#restoreNamedInstanceVariablesOf:toThoseOf:
	#restoreShapeOf:toThatOf:
	#restoreStateOf:toThatOf:
initializing	**#initialize**
	#initializeUndoMap

Class Method Types

instance creation	**#new**

Instance Methods

abort

Sets each object in the *undoMap* back to it's copy. Does this via #restoreStateOf:toThatOf:.

See also: #restoreStateOf:toThatOf:

begin

Initializes the *undoMap* to an empty dictionary, thus preparing it for the next “undo” session.

See also: #commit

commit

There is currently no difference between this method and #begin.

See also: #begin

### initialize

Calls #initializeUndoMap

See also: #initializeUndoMap

initializeUndoMap

Sets the *undoMap* instance variable to a new, empty *IdentityDictionary* instance.

instanceVariablesOf: *anObject* **do:** *aBlock*

There are no senders of this method. As near as I can tell, it is not currently being used. It is also marked as private, so in any case, you shouldn't need it.

aBlock is a one argument block. Each of the instance variables of *anObject* are passed to the block in turn. Following that, if *anObject* is an indexable object (like an *Array*, *String*, etc.), then what is found at each index is sent to *aBlock*, beginning at the first index.

isRegistered: *anObject*

For *anObject* to be registered, it will be found as a key in the *undoMap*. If it's not in the *undoMap*, then it is not registered.

Well, that's almost accurate. *anObject* might also be a proxy (an instance of *Proxy*) that has already been instantiated, thus the real object will be in the *undoMap* rather than the proxy. In this case, the real object is retrieved from the proxy, and then the real object is tested to see if it is in the *undoMap*.

Nils can't be registered, so if the argument *anObject* is nil, we know we need to return false. But it is slightly more complicated than that, because *anObject* might be a proxy whose real object is nil. We can handle both of these cases by just testing to see if the real object is nil, and if so, return false, and so that is what we do.

See also: #realObjectFor:

isShapeOf: *original* **differentThanThatOf:** *copy*

Private. Tests to see if the classes and basicSize of the *original* vs. *copy* are the same.

realObjectFor: *anObject*

anObject is the real object unless it is an instance of *Proxy* and has also been previously instantiated. In this case we ask the proxy for it's real value, and return that. Otherwise we just return *anObject*. Notice that this method does not force a proxy to instantiate itself.

See also: #isRegistered:

register: *anObject*

Make *anObject* be a member of the current transaction, if it is not already a member. Return *anObject* if we registered it with this call, or nil otherwise.

The process of registering means we make a shallow copy of *anObject*, and then do the following:
'undoMap at: realObject put: copy'.

This method also handles *anObject* just fine if it is a collection, by explicitly registering any internal structures of the collection as needed.

registeredObjectsDo: *aBlock*

aBlock is a one argument block. Each of the keys of the undoMap are passed to this block, one at a time.

requiresRegistrationFor: *anObject*

Answers whether or not *anObject* can be registered. For it to be registerable, it's real object must not be nil, and it must not already be registered.

Note that there is a little bit of code duplication between this method and #isRegistered: that probably should be factored out eventually.

See also: #isRegistered:

restoreIndexedInstanceVariablesOf: *original* **toThoseOf:** *copy*

Private. Does just what the method name says.

See also: #restoreStateOf:toThatOf:

restoreNamedInstanceVariablesOf: *original* **toThoseOf:** *copy*

Private. Does just what the method name says.

See also: #restoreStateOf:toThatOf:

restoreShapeOf: *original* **toThatOf:** *copy*

Private. Does a #basicNew on the class of the copy, then has the original become the copy. The #become: method swaps the object pointers, thus after this method is called, the *copy* is the old original, and the new *original* is a fresh, uninitialized instance of that same class.

See also: #restoreStateOf:toThatOf:

restoreStateOf: *original* **toThatOf:** *copy*

Private. If needed, restores the shape of the *original* (via #restoreShapeOf:toThatOf:). Then restores the values of everything found in *copy* back into the *original*.

shallowCopyOf: *anObject* **ifNotNeeded:** *aBlock*

Private. If *anObject* and a shallow copy of *anObject* are the exact same object, then evaluates *aBlock* and returns the result. Otherwise returns the shallow copy.

undoMap

Returns the *undoMap*.

Class Methods

new

Implemented as '^super new initialize'.

OraclePlatform

Inherits from: DatabasePlatform : Object

Class Description

This class is not particularly important for understanding the GLORP framework[13]. You will only use this class to create a login instance, and you can look at the PostgreSQLPlatform class documentation for an example of doing this.

See the *Login* and *DatabasePlatform* classes for more information.

13 Unless you are trying to create or extend support for other RDBMS's.

PostgreSQLPlatform

Inherits from: DatabasePlatform : Object

Class Description

For the reasons mentioned below, this class is not important for understanding the GLORP framework[14]. You will only use this class to create a login instance.

The support in DatabasePlatform and all of it's subclasses (including this one) is DDL-oriented (Data Definition Langauge, which is the part of SQL that deals with database schema creation and manipulation). Most of your DDL code is going to be for table creation, and I personally prefer to keep such code as external shell scripts, hence I personally don't see myself as using the DatabasePlatform class (or any of it's subclasses) at all, except to create a Login instance.

In the example below, we create a login instance for accessing the 'db' PostgreSQL database residing on the machine named 'host', and with user name of 'username' and password of 'password':

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

Once you have created your Login instance, you create a platform-specific DatabaseAccessor subclass instance thus:

```
accessor := DatabaseAccessor forLogin: aLogin.
```

Then you use the accessor instance for the rest of things, as described elsewhere.

14 Unless you are trying to create or extend support for other RDBMS's.

PSQLInt4DatabaseType

Inherits from: DatabaseType : Object

Class Description

This appears to be another one of those classes you can ignore.

The purpose of this class is to support the database-specific way of specifying four byte integer numbers. For PostgreSQL, a four byte integer is specified via 'int4', hence that is the value returned by the #typeString method of this class. In contrast, MS Access uses 'integer' to specify a four byte integer.

Creating a class for this simple functionality seems to me to be extreme overkill, but we'll see later.

PSQLSequenceDatabaseType

Inherits from: DatabaseType : Object

Class Description

This appears to be another one of those classes you can ignore.

The purpose of this class is to support the database-specific ways of generating sequence numbers.

PSQLTextDatabaseType

Inherits from: DatabaseType : Object

Class Description

This appears to be another one of those classes you can ignore.

The purpose of this class is to support the database-specific way of specifying text data in the database. For PostgreSQL, text data is specified via 'text', hence that is the value returned by the #typeString method of this class.

Creating a class for this simple functionality seems to me to be extreme overkill, but it does seem rather obvious that more is planned for this class, and similar classes, at some future time.

PSQLVarCharDatabaseType

Inherits from: VariableDatabaseType : DatabaseType : Object

Class Description

This appears to be another one of those classes you can ignore.

The purpose of this class is to support the database-specific ways of generating code for variable length character data within the database.

Session

Class Description

This class has been renamed to *GlorpSession* because of a name collision with the *Session* class of the PWS system in Squeak. Please see the section on *GlorpSession* for more information.

Tracing

Inherits from: Object

Class Description

This class doesn't seem to do much accept keep a collection of mapping expressions. Those mapping expressions in turn are added from a mapping class.

Instance Variables

base	An instance of *BaseExpression*
allTracings	A collection of mapping expressions

Instance Method Types

accessing	**#addTracing:**
	#base
	#base:
initialize	**#initialize**
querying	**#traceNodeSets**
	#tracesThrough:

Class Method Types

instance creation	**#new**

UnitOfWork

Inherits from: Object

Class Description

Takes responsibility for managing normal RDBMS transactions, but also integrates them into an undo mechanism. Also handles registering to GLORP so that they can be automatically updated.

This means that the *UnitOfWork* class must be the primary handler of database transactions. If you instead bypass the *UnitOfWork* functionality, thinking to instead go straight to lower layers to explicitly issue, say, transaction begins, commits or aborts (perhaps because you think all you need is a little bit of SQL executed, and not the entire GLORP framework), you will not be able to use the undo mechanism built into GLORP. Plus, since objects won't be registered to GLORP, you will not be able to use any of the automatic mechanisms of GLORP.

The lower layers would include any other mechanism that allows you to issue SQL straight to the datbase. This would include the EXDI layer (for VisualWorks), or straight to the `PGConnection>>execute:` code for PostgreSQL on Squeak, or even to other GLORP layers like `DatabaseAccessor>>doCommand:`. You should not try to manage transactions through any of those layers, but instead let the *UnitOfWork* class do it.

Instance Variables

session	An instance of *GlorpSession*, which can be thought of as the interface between your application and the RDBMS.
transaction	An ObjectTransaction, which can be considered a private class for the exclusive use of the *UnifOfWork* class. An ObjectTransaction takes responsibility for most of the undo mechanism, and is also the actual registrar for registering objects to GLORP.
deletedObjects	The actual objects that are going to be deleted at the next commit. This is different from the *deletePlan*, which is a collection of *DatabaseRows* built from each *deletedObject.*
newObjects	The collection of objects that have been registered with this *UnitOfWork.* Either that, or else I'm misuderstanding, because if this is what I just said, it really doesn't seem to be needed, because I could just ask the *transaction* for it's *undoMap* for this.
rowMap	A collection of RowMap's which were generated from the *commitPlan* objects or the *deletePlan* objects.
commitPlan	A collection of rows (instances of DatabaseRow) that need committing (writing) to the database.
deletePlan	A collection of rows (instances of DatabaseRow) that need deleting from the database.

Instance Method Types

accessing	**#correspondenceMap**
	#newObjects
	#session
	#session:
begin/commit/abort	**#abort**
	#begin
	#commit
	#createMementoRowMapFor:
	#createRowMapFor:
	#createRows
	#createRowsForCompleteWrites
	#createRowsForPartialWrites
	#isNewObject:
	#mementoObjects
	#postCommit
	#preCommit
	#registerTransitiveClosure
	#registeredObjects
	#rollback
deletion	**#delete:**
	#hasPendingDeletions
	#willDelete:
enumerating	**#registeredObjectsDo:**
	#rowsForTable:do:
initializing	**#initialize**
	#reinitialize
registering	**#isRegistered:**
	#register:
	#registerAsNew:
private	**#privateGetRowMap**
	#privateGetTransaction
	#sendPostWriteNotification
	#sendPreWriteNotification
private/mapping	**#addObject:toCacheKeyedBy:**
	#addToCommitPlan:
	#addToDeletePlan:
	#buildCommitPlan
	#readBackNewRowInformation
	#registerTransitiveClosureFrom:
	#updateSessionCache
	#updateSessionCacheFor:withRow:
	#writeRows

Class Method Types

instance creation	**#new**

Instance Methods

abort

Sends #reinitialize. This is the same thing as #begin does, which means that conceptually, a begin has an implicit abort, and visa-versa.

See also: #reinitialize

addObject: *anObject* **toCacheKeyedBy:** *key*

Private. Tells the *session* to put the *anObject* into the cache, at the requested *key*.

addToCommitPlan: *aRow*

Private. Adds *aRow* to the *commitPlan* collection.

addToDeletePlan: *aRow*

Private. Adds *aRow* to the *deletePlan* collection.

begin

Sends #reinitialize. This is the same thing as #abort does, which means that conceptually, a begin has an implicit abort, and visa-versa.

See also: #reinitialize

buildCommitPlan

Private. Initializes and populates the *commitPlan* and *deletePlan* collections. It queries the *GlorpSession* instance for information that it needs to populate them.

commit

Commits everything to the database.

correspondenceMap

Answers the *undoMap* of the transaction (which is a private instance of *ObjectTransaction*). The undoMap is an *IdentityDictionary* containing keys of original objects, and values of copies of those original objects before the original objects were changed.

Why is this now being called the *correspondenceMap*?

createMementoRowMapFor: *objects*

Create a rowmap for the objects in the argument whose state was already known. At some later time, we will subtract this from the rowmap of all known objects to get the rows that need to be written.

Off hand, I don't see why you would ever call this directly, as the public API is #createRows. Perhaps this method should be moved to 'private'?

See also: #createRows

createRowMapFor: *objects*

Create a rowmap for all of the objects in the argument, regardless of whether or not those objects were already known.

Off hand, I don't see why you would ever call this directly, as the public API is #createRows. Perhaps this method should be moved to 'private'?

See also: #createRows

createRows

Currently just sends #createRowsForPartialWrites.

I don't think you need to directly call this. The #preCommit method calls it, which in turn is called by #commit, which is the method that you would call.

See also: #createRowsForPartialWrites

createRowsForCompleteWrites

Not used. This is just a reference implementation. Use #createRowsForPartialWrites instead.

See also: #createRowsForPartialWrites

createRowsForPartialWrites

Creates needed rows in the cache, in advance of actually writing those rows to the database.

Off hand, I don't see why you would ever call this directly.

delete: *anObject*

Adds *anObject* to the *deletedObjects* collection.

Off hand, I don't see why you would ever call this directly.

hasPendingDeletions

Is *deletedObjects* empty?

initialize

Standard initialization stuff

isNewObject: *anObject*

Checks to see if *anObject* is in the *newObjects* collection, which is a collection of objects that have been registered with this *UnitOfWork*. What is the difference between what this returns, and what `#isRegistered:` returns?

isRegistered: *anObject*

Asks the *transaction* if *anObject* is registered.

See also: #isNewObject:

mementoObjects

Answers the correspondenceMap, which in turn is the *undoMap* of the transaction. The method has the following comment:

Warning: Excessive cleverness!!! The mementoObjects we want to iterate over are the values in the correspondenceMap dictionary. We were getting the values and returning them, but if all we need to do is iterate, then the dictionary itself works fine.

newObjects

Getter for the *newObjects* instance variable. See the instance variable comments for more information.

postCommit

Does stuff.

preCommit

Does stuff.

privateGetRowMap
privateGetTransaction

Private methods.

readBackNewRowInformation

Does stuff.

register: *anObject*

Does stuff.

registerAsNew: *anObject*

Does stuff.

registerTransitiveClosure

Does stuff.

registeredObjects

Does stuff.

registeredObjectsDo: *aBlock*

Does stuff.

reinitialize

Does stuff.

rollback

Does stuff.

rowsForTable: *aTable* **do:** *aBlock*

Does stuff.

sendPostWriteNotification

Does stuff.

sendPreWriteNotification

Does stuff.

session

Does stuff.

session: *anObject*

Does stuff.

updateSessionCache

Does stuff.

updateSessionCacheFor: *anObject* **withRow:** *aRow*

Does stuff.

willDelete: *anObject*

Does stuff.

writeRows

Does stuff.