Object Oriented Solutions for Structured Data Access

JOHN KARR
PHILADELPHIA LINUX USERS GROUP
MARCH 2014

What is a Database?

- A Database is a mechanism for storing and retrieving data on a computer system.
- A database is characterized by the ability to selectively and quickly retrieve data from it.
- There are two principal Tribes of DataBase: Relational and Non-Relational.

Non-Relational DataBases

- The classic non-relational databases are DBM and BerkeleyDB, these key value databases while much faster than simple file system archival, still need to search every record to find 'Red House‘ unless it happens to be the key.
- Modern Non-Relational Databases, often referred to as NoSQL, such as Hadoop, MongoDB, Cassandra, and CouchDB, are much much better at selective data retrieval. They've become popular due to simplicity and speed.

Relational Databases

- What differentiates a Relational Database is that it maintains relations between objects and usually has mechanisms to enforce data integrity.
- During the early days of the PC revolution dBASE was ubiquitous, but ultimately lost out to SQL for Multi-User applications and Access and FileMaker for single user applications.

SQL was developed at IBM by Donald D. Chamberlin, Donald C. Messerly, and Raymond F. Boyce in the early 1970s. The acronym stands for Structured Query Language.

- Data Definition Language used to define and manipulate data structures.
- Data Manipulation Language for querying and manipulating data.
- the ACID principle:

    Atomicity

    Consistency

    Isolation

    Durability

ACID

- **Atomocity**: No other operations will be affected by a write until it is complete and entire.
- **Consistency**: All actions must leave the database in a state consistent with its rules. Aborted transactions should have no effect.
- **Isolation**: One transaction must not affect another.
- **Durability**: Once a transaction is committed it must persist even through crashes.

MongoDB Example

```
#!/usr/bin/ruby
require 'mongo'
include Mongo
mongo_client = MongoClient.new("localhost",
27017)
puts 'List of databases'
mongo_client.database_info.each {
                    |info|  puts info.inspect }
mydb =
  MongoClient.new("localhost", 27017).db("mydb")
mycoll = mydb.collection("testCollection")
doc = {
      "name" => "MongoDB",
      "type" => "database",
      "count" => 1,
      "info" => {"x" => 203, "y" => '102'}
}
mycoll.insert(doc)
5.times { |i|  mycoll.insert("i" => i) }
puts ‘display the records'
mycoll.find.each { |row|  puts row.inspect }
mycoll.remove
```

OUTPUT

List of databases
["mydb", 218103808]
["local", 83886080]
insert some records and display all of the records
{"_id"=>BSON::ObjectId('5313978c23e9694081000001'), "name"=>"MongoDB", "type"=>"database", "count"=>1, "info"=>{"x"=>203, "y"=>"102"}}
{"_id"=>BSON::ObjectId('5313978c23e9694081000002'), "i"=>0}
{"_id"=>BSON::ObjectId('5313978c23e9694081000003'), "i"=>1}
{"_id"=>BSON::ObjectId('5313978c23e9694081000004'), "i"=>2}
{"_id"=>BSON::ObjectId('5313978c23e9694081000005'), "i"=>3}
{"_id"=>BSON::ObjectId('5313978c23e9694081000006'), "i"=>4}

A similar example in SQL (Postgres)

```
#SQL

CREATE TABLE plug
(   i integer,
 id serial NOT NULL,
 info character varying,
 CONSTRAINT pkey_plug_id PRIMARY KEY (id) ) ;
GRANT ALL ON TABLE plug TO test;
GRANT ALL ON TABLE plug_id_seq TO test;

#RUBY
require 'pg'

jsondata = <<JSOND
'{
"name" : "MongoDB",
"type" : "database",
"count": 1,
"info" : {"x" : 203, "y" : 10 }
}'
JSOND
```

```
conn = PG::Connection.open(
  :dbname => 'test',
  :host => 'test',
  :user => 'test',
  :password => 'test')

conn.exec( "insert into plug ( info ) values (
#{jsondata} )" )
5.times {   |i|
conn.exec( "insert into plug ( i ) values ( #{i} )" )
}
res = conn.exec( 'select * from plug' )
res.each do |row|
row.each do |column|
print "#{column}\t"
end
print "\n"
end

conn.exec( 'truncate plug')
```

The Results

["i", nil] ["id", "79"] ["info", "{\n \"name\" : \"MongoDB\",\n \"type\" : \"database\",\n \"count\": 1,\n \"info\" : {\"x\" : 203, \"y\" : 10 }\n}"]

["i", "0"] ["id", "80"] ["info", nil]

["i", "1"] ["id", "81"] ["info", nil]

["i", "2"] ["id", "82"] ["info", nil]

["i", "3"] ["id", "83"] ["info", nil]

["i", "4"] ["id", "84"] ["info", nil]

Why Mongo is better

- Create objects on the fly.
- Collections accept dissimilar records.
- Although Mongo supports requiring authentication of users, it was not mandatory to grant the test account explicit rights to the table and the sequence.
- Generally much more concise data manipulation syntax.
- Fast retrieval without defining and managing indexes.

Why SQL is better

- Imposes structure on data.
- Extensible through user defined functions (Stored Procedures). This allows Business Intelligence to be implemented in the DBMS.
- ACID.
- While Mongo gave us a primary key which we had to define in SQL, SQL will allow almost any conceivable sort of data constraint.
- SQL can combine multiple actions into a single transaction with changes only committed on success of all actions.
- SQL supports triggers which execute when an event occurs.

Structured Data

- This talk is about structured data, so the rest of the talk is going to be about SQL. I'll be using Postgres.
- SQL was designed in the 1970s and while very powerful it is also verbose and repetitious.

- DBI first appeared in 1992 to allow Perl programs to access SQL data.
- Because Perl's DBI is kind of cumbersome at times I use a DBI Wrapper, DBIx::Simple. DBIx::Simple is a very thin sugary coating on DBI and some auxiliary Modules frequently used in conjunction with it.

Bind Values

SQL is very verbose and repetitious and does not natively follow an object oriented approach that would allow programmers to implement a DRY (Don't Repeat Yourself) approach to writing code.

If we use just SQL our code is going to be littered with HERE Documents, or whatever the equivalent is in your language of choice. To facilitate reuse of Queries most SQL libraries support 'Bind Variables', when there are a lot of variables, managing them becomes its' own issue.

```
my $query = <<QUERY ;
INSERT (shell, filling, cheese, lettuce, sauce, peppers )
INTO taco
VALUES( ?,?,?,?,?,? )
QUERY

$DB->query( $query, 'crunchy', 'beef', 'TRUE', 'TRUE', 'medium', 'FALSE' );
```

In the late 1990s there began to appear SQL abstraction tools, such as DBIx::Abstract (now officially deprecated in favor of SQL::Abstract).

With SQL::Abstract you can generate the more common sql statements from an Associative Array. Since programmers are often working with data in Associative Arrays, this at least saves writing a lot of SQL in here documents.

```
my %taco_details = () ;
if ( $soft ) { $taco_details{shell} = 'soft' }
else { $taco_details{shell} = 'crunchy' }
...
$DB->insert( 'taco', \%taco_details );
```

(without DBIx::Simple the following would generate variables for dbi.)

```
my ( $stmt, @bind ) = $abstract->insert($table, \%fieldvals) ;
```

ORM

- The next step in the evolution of working with SQL were Object Relational Mappers, ORM.
- Generally the goal of ORM is to provide an object oriented abstraction layer above your database so that you can work with your data as a native object in your language. A secondary goal followed by most ORMs is to provide database independence.
- In 2003 Martin Fowler in Patterns of Enterprise Application Architecture defined two patterns that summarize most ORM implementations:

P of EAA: Active Record

www.martinfowler.com/eaaCatalog/activeRecord.html

Most Visited Getting Started Latest Headlines

MARTIN FOWLER

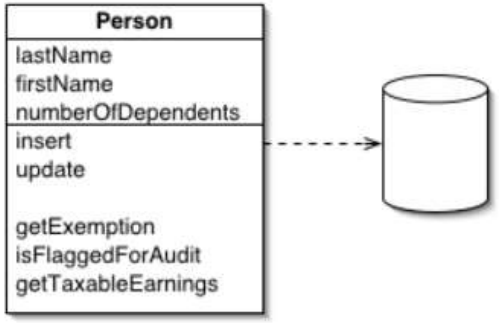
Intro Design Agile Refactoring NoSQL DSL Delivery About Me ThoughtWorks

| P of EAA Catalog |

Active Record

An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

For a full description see P of EAA page **160**



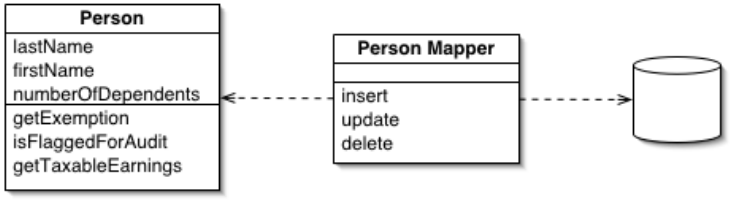
An object carries both data and behavior. Much of this data is persistent and needs to be stored in a database. Active Record uses the most obvious approach, putting data access logic in the domain object. This way all people know how to read and write their data to and from the database.

| P of EAA Catalog |

Data Mapper

A layer of Mappers (473) that moves data between objects and a database while keeping them independent of each other and the mapper itself.

For a full description see P of EAA page **165**



Objects and relational databases have different mechanisms for structuring data. Many parts of an object, such as collections and inheritance, aren't present in relational databases. When you build an object model with a lot of business logic it's valuable to use these mechanisms to better organize the data and the behavior that goes with it. Doing so leads to variant schemas; that is, the object schema and the relational schema don't match up.

You still need to transfer data between the two schemas, and this data transfer becomes a complexity in its own right. If the in-memory objects know about the relational database structure, changes in one tend to ripple to the other.

The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With Data Mapper the in-memory objects needn't know even that there's a database present; they need no SQL interface code, and certainly no knowledge of the database schema. (The database schema is always ignorant of the objects that use it.) Since it's a form of Mapper (473), Data Mapper itself is even unknown to the domain layer.

Ruby Activbe Record

- The most popular and well known ORM is probably Ruby's Active Record.
- Here's an example stolen from Michael Hartl's Ruby on Rails Tutorial

```
>> User.find_by_email("mhartl@example.com")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2013-03-11 00:57:46", updated_at: "2013-03-11 00:57:46">
```

DBIx::Class

- n Perl the dominant ORM is DBIx::Class. There is some confusion as to whether DBIx::Class is an Active Record implementation. Yes and No. DBIx::Class was written completely independent of Ruby's Active Record (in fact it was written as a replacement for one of the earliest ORMs, Class::DBI), and I don't believe that its' author Matt Trout had read Fowler's book.
- DBIx::Class does generally follow the Active Record Pattern described by Fowler.
- This is the preceeding example in DBIx::Class.

```
my $result = $schema->resultset('User')->find(
    { email => "mhartl@example.com" } );
say $result->id ;
say $result->name ;
say $result->email ;
```

Advantages and Disadvantages of ORM

- Philosophically, ORMs approach the DataBase as a repository of the application's data.
- Conversely a DBA might see the Database itself as the object of importance, and the program as a means to access the Data.

  If your view is DataCentric then most ORMs are written backwards.
- Database Independence also means forgoing useful Database specific features.

The Next Generation

- Both DBIx::Class and Active Record are about a decade old now.
- In that time the hot development is the new generation of NoSQL.
- There are ORMs for NoSQL, but a lot of their allure is that they are more compatible with modern programming from the beginning.

My Project

- Vaporware: The PostgreSQL DataObject (PgDO)
- Perl Object Oriented Extension to Postgres.
- Can use all Postgres features because no other DBMS is supported.
- Encourage SQL when appropriate, avoid mundane SQL.
- DOML (Data Object Markup Language) to concisely represent database objects.
- Extension to DOML to permit providing formatted data to the View in Model View Controller.