AndyPatterns

Object Relational Mapping Pattern - using SQLObject

Relational databases and OO memory models use different paradigms. Mapping between one and the other is often necessary and ends up being fiddly work so an auto-mapping tool like SQLObject (python framework) can help enormously.

How it works

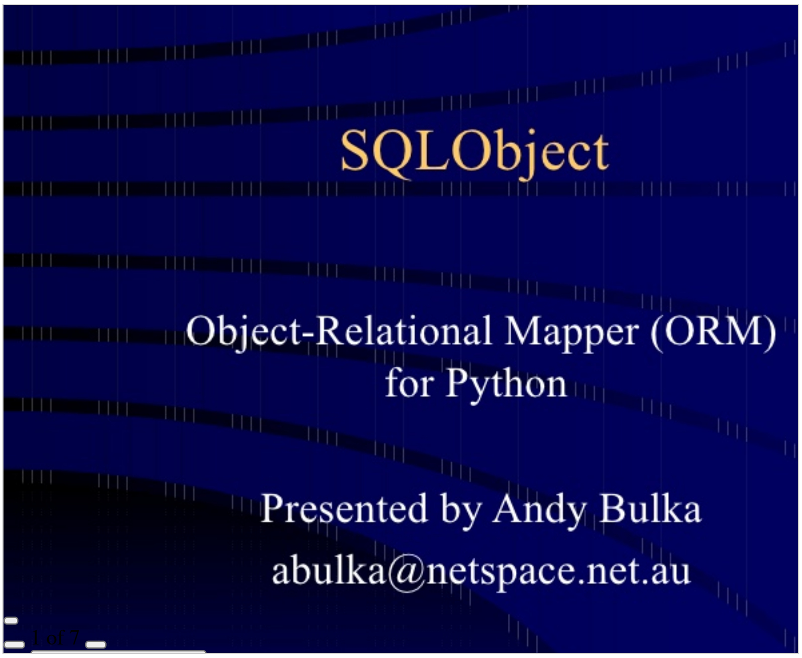
Let's face it - a lot of developers prefer to just code classes - that's what OO is all about. Having then to persist those classes into tables creates all sorts of fiddly mapping code. If you don't need to persist into tables - fine. But if you do need to persist into tables, one way to reduce your workload is to use a tool like SQLObject - you simply write your code in a certain way and the tables are automatically maintained.

- Each class becomes a table.
- Each attribute on a class is a column in a table.

The ORM tool automatically causes the setting of attributes on objects to have a wonderful side effect: the values are written to the database table. And conversely, pulling information out from your object triggers the appropriateSQL calls to retrieve the information from the table. **You don't have to deal with sql syntax yourself**. However you can take advantage of the fact that you now have a sql backend, to make sql-like queries - thus you can save coding effort. You have all the power of SQL at your disposal now, you might as well use take advantage of it! You don't have to use precise SQL syntax but you need to use something very much like it E.g. in SQLObject you would write things like:

```
ps = Person.select("""address.postcode LIKE '3186'""",
                   clauseTables=['address'])
print list(ps)
```

Finally, ORM (object relational mapping) tools usually let you choose what style of database to use with a switch - e.g. develop on sql-lite and deploy on mysql - all without changing any of your code.



SQLObject

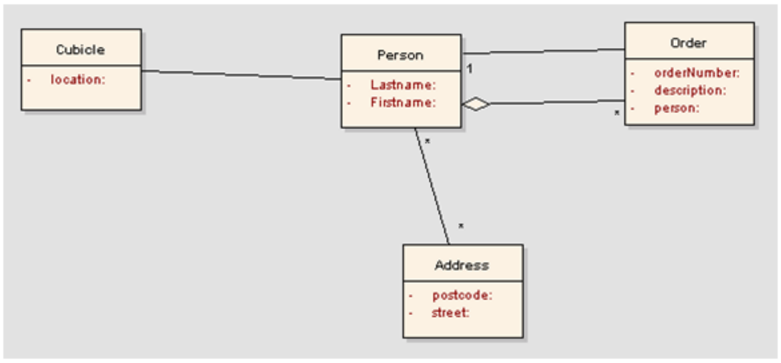
Object-Relational Mapper (ORM)
for Python

Presented by Andy Bulka
abulka@netspace.net.au

Object Relational Mapping Pattern - using Sql Object from **tcab22**

Code Example - Before using SQLObject

This is an example of some Person, Cubicle and Order classes. Persons live in Cubicles and can place one or more orders. No database activity is involved here - its all just regular python code with pointers and lists.



The UML used in our example

```
# BEFORE the use of a database

class Cubicle:
    def __init__(self, location="unknown"):
        self.location = location
        self.occupant = None

    def SetOccupant(self, person):
        # Evict any previous occupant
        if self.occupant:
            self.occupant.cubicle = None
        self.occupant = person  # New occupant wired in
        person.cubicle = self   # back pointer

class Person:
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname
        self.cubicle = None
        self.address = None
        self.orders = []
        self.addresses = []

    def AddOrder(self, order):
        self.orders.append(order)  # one to many
        order.person = self        # back pointer

    def SetAddress(self, address):
        self.addresses.append(address) # many to many
        address.residents.append(self) # back pointer (note the 'append' cos many to many)

class Order:
    def __init__(self, orderNumber, description):
        self.orderNumber = orderNumber
        self.description = description

class Address:
    def __init__(self, street, suburb, postcode):
        self.street = street
        self.suburb = suburb
        self.postcode = postcode
        self.residents = []


# Test One to one

cubicle1 = Cubicle("North Wing D4")
tom = Person(firstname="Tom", lastname="Jones")
cubicle1.SetOccupant(tom)
assert cubicle1.occupant == tom

# Test One to many

o1 = Order(orderNumber="12345", description="new ipaq")
o2 = Order(orderNumber="12346", description="new ipod")
tom.AddOrder(o1)
tom.AddOrder(o2)
assert len(tom.orders) == 2
assert o1 in tom.orders
assert o2 in tom.orders

# Test Many to many

angelina = Person(firstname="Angelina", lastname="Jolie")
a1 = Address("Fox Studios", "California", 3186) # tom and angelina both work here
a2 = Address("Brads Place", "Manhattan", 40004)
```

```
angelina.SetAddress(a1)
angelina.SetAddress(a2)
tom.SetAddress(a1)
assert a1 in angelina.addresses
assert angelina in a1.residents
assert angelina in a2.residents
assert tom in a1.residents
assert tom not in a2.residents

################## Now do some more complex manipulations #########

# Move angelina into the North Wing D4 cubicle and
# move Tom into a new cubicle

cubicle1.SetOccupant(angelina)
assert cubicle1.occupant == angelina
assert tom.cubicle == None

cubicle2 = Cubicle("West Wing D5")
cubicle2.SetOccupant(tom)
assert tom.cubicle == cubicle2

print 'Done!'
```

Output:

```
Done!
```

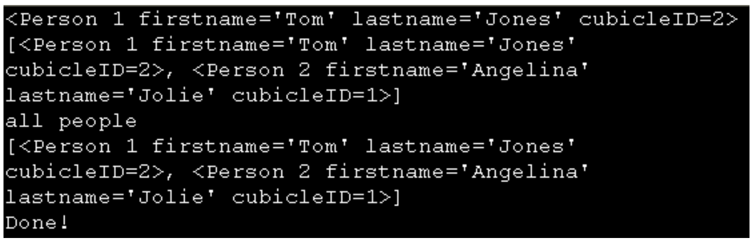
Code - After SQLObject

Now for the fascinating transformation. We refactor the code to use SQLObject and we end up with the same logic, but as a bonus we have persistence in a sql database. And its not so hard either!

```
# AFTER - the database version - notice we import sqlobject

from sqlobject import *
from sqlobject.sqlite import builder; SQLiteConnection = builder()
conn = SQLiteConnection('person.db', debug=False)

class Cubicle(SQLObject):
    _connection = conn
    location = StringCol(length=20, default="unknown")
    occupant = ForeignKey('Person', default=None)

    def SetOccupant(self, person):
        # Evict any previous occupant
        if self.occupant:
            self.occupant.cubicle = None
        self.occupant = person  # New occupant wired in
        person.cubicle = self   # back pointer

class Person(SQLObject):
    _connection = conn
    firstname = StringCol(length=20)
    lastname = StringCol(length=20)
    cubicle = ForeignKey('Cubicle', default=None)
    orders = MultipleJoin('GiftOrder')
    addresses = RelatedJoin('Address')

    def AddOrder(self, giftOrder):
        #self.orders.append(giftOrder)  # one to many # SQL OBJECT doesn't need this
        giftOrder.person = self        # back pointer ** becomes the primary info sqlobject goes on

    def SetAddress(self, address):
        #self.addresses.append(address) # many to many # SQL OBJECT doesn't need this
        #address.residents.append(self) # back pointer (note the 'append' cos many to many) # SQL OBJECT doesn't
        address.addPerson(self)  # SQLobject created this "addWHATEVER" method for us

class GiftOrder(SQLObject):
    _connection = conn
    orderNumber = IntCol()
    description = StringCol()
    person = ForeignKey('Person', default=None)

class Address(SQLObject):
    _connection = conn
    street = StringCol(length=20)
    suburb = StringCol(length=20)
    postcode = StringCol(length=20)
    residents = RelatedJoin('Person')
    #def _init(self):
    #    SQLObject._init(self, *args, **kw)
    #    self.postcodesDict = {'2323':'Brighton','22222':'Werribee'}
```

```
Cubicle.dropTable(True)
Cubicle.createTable()
Person.dropTable(True)
Person.createTable()
GiftOrder.dropTable(True)
GiftOrder.createTable()
Address.dropTable(True)
Address.createTable()


# Test One to one

cubicle1 = Cubicle(location="North Wing D4")
tom = Person(firstname="Tom", lastname="Jones")
cubicle1.SetOccupant(tom)
assert cubicle1.occupant == tom

# Test One to many

o1 = GiftOrder(orderNumber=12345, description="new ipaq")
o2 = GiftOrder(orderNumber=12346, description="new ipod")
tom.AddOrder(o1)
tom.AddOrder(o2)
assert len(tom.orders) == 2
assert o1 in tom.orders
assert o2 in tom.orders

# Test Many to many

angelina = Person(firstname="Angelina", lastname="Jolie")
a1 = Address(street="Fox Studios", suburb="California", postcode="3186") # tom and angelina both work here
a2 = Address(street="Brads Place", suburb="Manhattan", postcode="40004")

angelina.SetAddress(a1)
angelina.SetAddress(a2)
tom.SetAddress(a1)
assert a1 in angelina.addresses
assert angelina in a1.residents
assert angelina in a2.residents
assert tom in a1.residents
assert tom not in a2.residents

################# Now do some more complex manipulations #########

# Move angelina into the North Wing D4 cubicle and
# move Tom into a new cubicle

cubicle1.SetOccupant(angelina)
assert cubicle1.occupant == angelina
assert tom.cubicle == None



cubicle2 = Cubicle(location="West Wing D5")
cubicle2.SetOccupant(tom)
assert tom.cubicle == cubicle2

# Now SQLOBJECT lets us do other magic things, that leverage relational db technology
p = Person.get(1)
print p

#ps = Person.select(Person.q.firstName=="John")
#print list(ps)

#ps = Person.select("""address.id = person.id AND
#                          address.postcode LIKE '40004%'""",
#                       clauseTables=['address'])
ps = Person.select("""address.postcode LIKE '3186'""",
                      clauseTables=['address'])
print list(ps)

print 'all people'
ps = Person.select()
print list(ps)

print 'Done!'
```

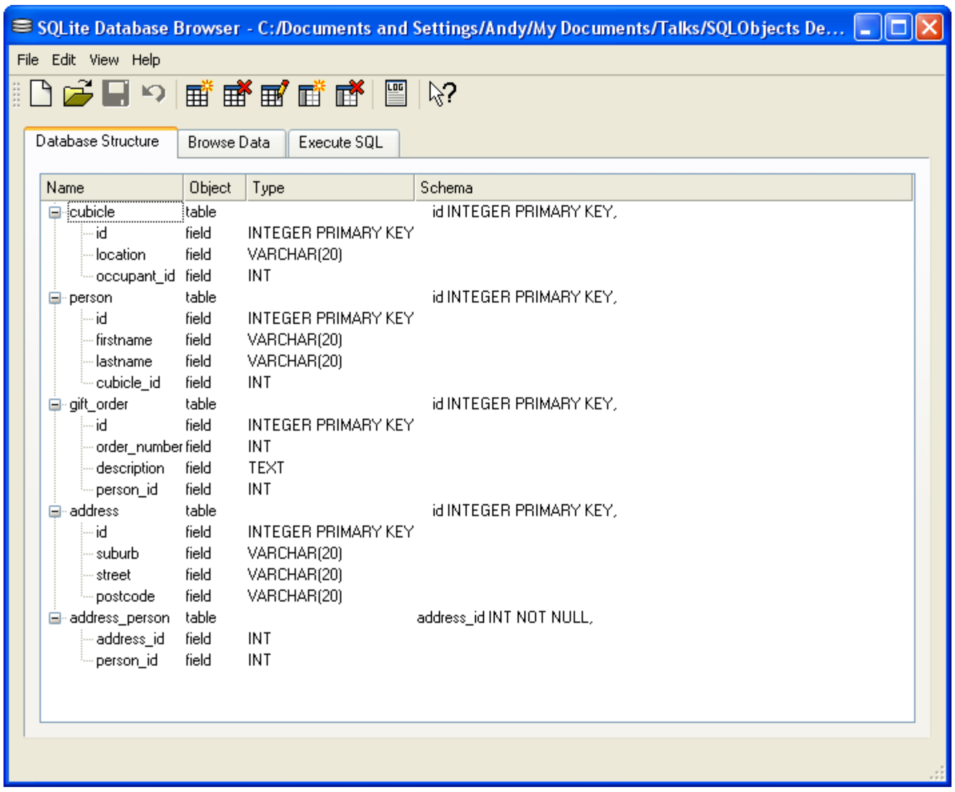
Output:



```
<Person 1 firstname='Tom' lastname='Jones' cubicleID=2>
[<Person 1 firstname='Tom' lastname='Jones'
cubicleID=2>, <Person 2 firstname='Angelina'
lastname='Jolie' cubicleID=1>]
all people
[<Person 1 firstname='Tom' lastname='Jones'
cubicleID=2>, <Person 2 firstname='Angelina'
lastname='Jolie' cubicleID=1>]
Done!
```

What sort of tables are created?

You can browse the resulting database tables (in this case sqllite tables) using the SQLite Database Browser. Here are some screenshots of the tables we have created:


SQLite Database Browser - C:/Documents and Settings/Andy/My Documents/Talks/SQLObjects De...
File
Edit
View
Help
Database Structure
Browse Data
Execute SQL

Name	Object	Type	Schema
cubicle	table		id INTEGER PRIMARY KEY,
id	field	INTEGER PRIMARY KEY	
location	field	VARCHAR(20)	
occupant_id	field	INT	
person	table		id INTEGER PRIMARY KEY,
id	field	INTEGER PRIMARY KEY	
firstname	field	VARCHAR(20)	
lastname	field	VARCHAR(20)	
cubicle_id	field	INT	
gift_order	table		id INTEGER PRIMARY KEY,
id	field	INTEGER PRIMARY KEY	
order_number	field	INT	
description	field	TEXT	
person_id	field	INT	
address	table		id INTEGER PRIMARY KEY,
id	field	INTEGER PRIMARY KEY	
suburb	field	VARCHAR(20)	
street	field	VARCHAR(20)	
postcode	field	VARCHAR(20)	
address_person	table		address_id INT NOT NULL,
address_id	field	INT	
person_id	field	INT	

this is the table structure

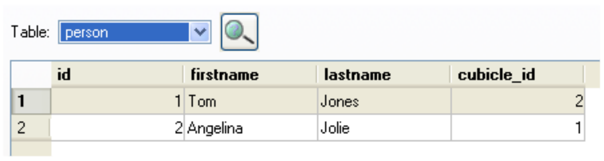


Table: person

	id	firstname	lastname	cubicle_id
1	1	Tom	Jones	2
2	2	Angelina	Jolie	1

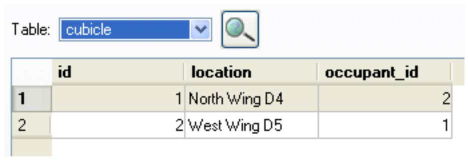


Table: cubicle

	id	location	occupant_id
1	1	North Wing D4	2
2	2	West Wing D5	1

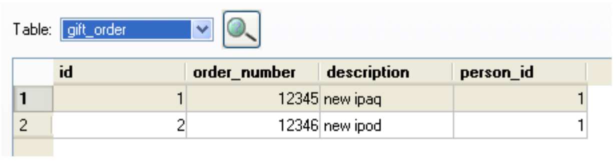


Table: gift_order

	id	order_number	description	person_id
1	1	12345	new ipaq	1
2	2	12346	new ipod	1

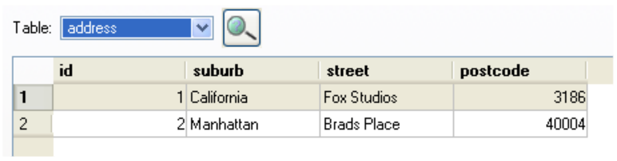


Table: address

	id	suburb	street	postcode
1	1	California	Fox Studios	3186
2	2	Manhattan	Brads Place	40004

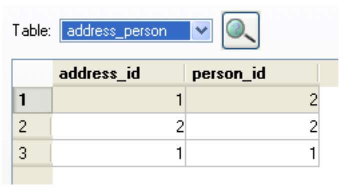


Table: address_person

	address_id	person_id
1	1	2
2	2	2
3	1	1

above is the data that was created by the code

Alternatives

You may also be interested in Sqlalchemy which seems to be getting a fair bit of mindshare. Its similar to SQLObject and is for python. Then there are ORM frameworks for java like hibernate and many others. Just google for ORM. And if you are into .NET then the "low hanging fruit" solution of LINQ is worth looking at.

The point of this article has been to get you to see the essence of what an ORM does, with the minimal amount of code.