AndyPatterns

Relationship Manager - Discussion

Before you read this discussion...

Remember that the *essential intent* of the Relationship Manager design pattern is that relationships are managed by something smart and central rather than distributed across millions of individual classes, with their individual implementations. The classes using RM still contain the business logic - they just call out to the RM to do all the mundane dirty work of adding and removing pointers etc. I think in the end there are similarities between RM and a database of relationships. The API of RM is just a sort of SQL.

RM is not about caching - though this is becomes an important issue, as repeated calls to look up a relationship can be slower than looking up a pointer. The latest C# implementation has some built in caching that helps improve performance.

Andy Bulka writes:

Hi again Peter and Nat (and now Don and James),

I've had some further thoughts about Relationship Manager Pattern (RM) which - in my mind - has escalated into a broader debate about the impedance mismatch between databases and objects in memory, and to the fascinating similarity between an in-memory RM and a disk based relational database.

If you are not interested in this post, then please safely ignore - and aplogies for clogging the airwaves. I'm taking this as an opportunity to document my recent thoughts here - I would, however be most interested in any thoughts and feedback on this stuff. I'll try to be as succinct as I can:

PRELUDE: RM (Relationship Manager) pattern describes an in-memory, mediator like object which stores the relationships between objects. So rather than implementing pointers and TLists and backpointers inside each class, we have the classes call the API of a central, in-memory RM.

The big benefit of this solution

Your classes should now implement relationships with simple one line calls e.g.

```
class Customer:
    def AddOrder(self, order):
        RelMgr.AddRelationship( From=self, To=order,
                                        RelId=CUSTORD )
```

rather than owning and later freeing a TList and more horribly, maintaining backpointers from Order objects back to their Customer, and keeping all this stuff in sync, with deletion notification, persistence issues etc. Yuk. Fast and we do it every day, but lots of stuff to keep track of vs. the alternative approach of simple *one line calls* and no local data/Tlists/pointers to manage.

1. **OBSERVATION**: RM stores/mediates the *relationships* between objects, but does not mediate the *behaviour* between those objects (behaviour == methods of the classes). Behaviour remains localised in the methods of the business object classes / classes of your model. Thus RM is more like a database of relationships rather than a strict mediator (which apparently according to GOF, mediates *behaviour*). Thanks to Nat for this point.

2. **BENEFIT**: By keeping the behaviour *out* of the RM (and thus preventing the RM from being a strict Mediator Pattern - mediating behaviour), the benefit is that classes still look like those in traditional OO designs - classes still have methods which gives each class its behaviour. The RM is thus simply a convenient way to *implement* relationship wiring, and is not a whole new way of designing classes.

** speculation begins .... **

3. **A BRAIN WAVE & SOME HISTORY**: The difference between Relational Databases and OO designs is the addition of behaviour to the classes. DB tables don't have behaviour (most people say this is a deliberate and good thing) - classes in memory DO have behaviour/methods.

NOTE: Hence is born the impedance mismatch, since 'dead behaviourless data' in RDMS's needs to be mapped to equivalent classes/objects in memory which *do* have behaviour.

Because I have so far repeatedly pointed out that RM takes great pains NOT to take on the responsibility of mediating behaviour, then RM is conceptually like a relational database, which too, by design, refuses to take on the responsibility of modelling behaviour.

Thus both RM and relational DB's refuse to model behaviour - for different reasons (and lets forget about stored procedures and triggers for the moment...). RM refuses to house behaviour because it exists for a different intent - to help manage wiring relationships between objects in memory. DB's refuse to house behaviour so that many varied pieces of software can behave differently yet operate with the same cold hard standard database entities.

4. **SPECULATION**: If we look at the [variation/version of] RM which also stores the objects/entities as well as the relationships - then the RM simply becomes an in-memory relational database!! But because it is in memory, it can refer to and store *real objects* (with their behaviour) and the relationships between real objects.

5. **BENEFIT**: An in-memory relational database (RM) lets us use relational language e.g. SQL-like statements or an equivalent function call API to do the hard work of looking after the wiring and querying of the relationship between objects. ARGUABLY the traditional use of pointers, back-pointers and TLists etc. is really trying to do similar work to that of a relational database, and this is reinventing the wheel. Sure, pointers and TLists are efficient but they are also painstaking since pointers and TLists have to be manually written and debugged for every class. Why reinvent the relational database wheel with buggy pointers - use an efficiently implemented, in-memory relational database / Relationship Manager instead.

NOTE: I'm not suggesting throwing away pointers and TLists etc. for everything you program - just throw them away for the formal relationships between your business objects in your OO model. Though you could have another/[the same?] RM that looked after other more mundane wirings in your software...e.g. observer relationships, etc. ;-)

6. **PERSISTENCE IS EASY**: An in memory database like RM still needs to be persisted, which is usually easy if you stream it all out. If RM is implemented as a TComponent, then streaming is trivial, and Delphi's pointer referencing reconstruction will magically recreate the RM with all the objects and relationships in memory (each relationship is a TComponent owned by RM, each business object is a descendant of TComponent and also owned by RM). Python is similarly easy to stream.

7. **PERSISTENCE FOR SMALL DATABASES IS A LITTLE HARDER** - If we want to store the RM as a traditional disk based DB, then we have the mapping problem / impedance mismatch problem all over again. We would have to read the entire DB into memory (into our RM) and later back out again. **SOLUTION**: Invent an algorithm which uses a combination of a schema of some sort (plus perhaps runtime type information) to do the mapping automagically. Or code it by hand. Furthermore, if you want the resiliency of writing to DB often (not just when the user consciously saves everything) then you get even further into the traditional complexities of 'dirty' flags and a complex persistence layer.

8. **PERISTANCE FOR LARGE DATABASES IS HARD**: - When you can't or don't want to read all the DB into a RM in memory, then a simple RM fails. When you have multiple users and client server architectures, then simple RM fails.

Summary

Congratulations if you have gotten this far!

RM is a useful class. I guess RM is ideal when no external traditional DB representation is required. RM can also be used with DB's that can be read entirely into memory, though traditional persistence layers are still then required.

If you like it, RM is a useful thing to have in memory - but it doesn't solve the persistence layer to DB problem, though it greatly simplifies the persistence to stream problem.

The fact that an in-memory RM is so similar to a disk based DB is fascinating to me - implementing wiring between objects in memory is done by a RM just like the wiring between objects [relationships] is maintained in a disk based DB.

-Andy

At the last melb patterns meeting, Nat was presenting Mediator pattern. I gave an impromptu sketch of the Relationship Manager as an example of a Mediator. Later that evening, over a beer, Nat suggested that RM wasn't strictly a mediator since the collegue objects still 'know about' each other. A mediator like a Delphi form on the other hand ties components together *without* the colleague components (e.g. a TButton and a TEdit) knowing about each other.

I guess I would respond that in Relationship Manager (RM), even though the colleague objects admittedly DO know about each other (e.g. a customer knows about its orders etc.) they DO NOT know the nitty details of how to get at each other (they instead as the RM). Thus even though there is not 'total and ultimate' mediation going on - there is nevertheless mediation of the details of how to refer to each other. Its still an important level of mediation, even if it aint total.

> Quoting Peter Hinrichsen :
>
> > Hello Andy,
> >
> > Your Relationship Manager has been gaining some popularity among the
> > tiOPF users. http://www.techinsite.com.au/tiOPF/MailingList.htm
> > Here is some of the chat.
> >
> > Rgs,
> >
> > Peter.
> >
> > >Hi Martin
> > >
> > >The Relationship Manager is intriguing. An immediate benefit from
> using it would be simplication of a querying language - I've been struggling
> for some time over the issue of how you'd implement something like OCL
> over tiOPF without writing a heap of visitors. This might be the way to
> support it.
> > >
> > >
> > >Cheers
> > >Tom
> > >
> > >----- Original Message -----
> > >From:

> > >To:
> > >Sent: Thursday, March 14, 2002 11:59 PM

> > >Subject: tiOPF-tiOPF question
> > >
> > >
> > >> Hello,
> > >>
> > >> here are some questions:
> > >>
> > >> Are there any ideas to use a Relationship Manager as discussed at
> > >> http://www.atug.com/andypatterns/rm.htm to decouple the
> > >> dependences betweenpersistent objects?
> > >>
> > >> Where is an entry point to separate the presentation-layer from
> > >> the BOM anddo some interfacing like COM?
> > >>
> > >> The previous EMail with a decorator araound tPerObjList is a very
> > >> good idea, because there are some other points to optimize: Loading the list
> > >> only as an integerlist of OIDs internally and do so, as when the objects
> > >> completlyloaded, and really load objects completely later on demand.
> > >>
> > >> Regards,
> > >>
> > >>
> > >> Martin

Further Musings on Relationship Manager and Persistence

by Andy Bulka (not emailed to anyone)

On using a real relational database as a Relationship Manager

I would love to somehow substitute something like Interbase/Firebird/MySQL for my RM, but the problem is DB's don't store real pointers/memory objects. And even if they did, memory pointers always change wheras object ID's in a DB don't.

> LATER THAT NIGHT, I GOT WHAT I WISHED FOR!! Later that evening I discovered **PyDO** which seems to be just this sort of thing!! See ensuing discussion, below >>

Perhaps this is the problem *Object Oriented* Databases solve - allowing you to have an 'in-memory' DB (a relationship manager :-) where you can have real objects and pointers. The downside is that OO Databases are not traditional relational DB's and their file formats are unique.

On table ID's vs. Object pointers

What about holding a separate mapping/dictionary, for each user, of memory objects to table/ID row.

Actually in traditional OO to DB persistence layers, memory objects have an attribute 'ID' which points to table/ID. Keeping all this in a separate mappingdictionary might be inefficient, but it might give us some benefits I can't quite think of right now.

An ID is an object address, and a foreign key is a pointer. So when we insert a new field into a database that has an auto-increment integer primary key, we often ask: what was the ID just allocated? This is the same as saying, what was the pointer to the newly created object, as we do when we create a new instance of a class.

Discussion with James #1

Quoting James Hutton Jhutton@au.mediacommand.com:

James: Elimination of backpointers strikes me as the most convincing benefit.

Andy: Yes, in fact the chore and bugginess of maintaining backpointers triggered my investigations into this area in the first place :-)

James: Why bother persisting to an RDB? As you say, it's not suitable for large amounts of data or multi-user situations, which are surely two of the main reasons you'd use an RDB in the first place. If you just persisted to a vanilla disk file, wouldn't life be easier?

Andy: Yep - I regularly stream RM to disk in both Delphi and Python. I haven't yet mapped a RM to a DB yet.

James: Suppose you have (A, B) and (B, C). Then by implication you have (A, > C). An RDB can show you that relationship. What about the RM?

Andy: A slightly smarter RM could deduce these implied relationships - though I guess you would have to decide whether you wanted your RM to be making these deductions, or only storing the wirings that you explicitly add.

Discovery of PyDO - another example of RM?

Late Last night I (Andy) was investigating object to relational database mappers. In one of the cleaner, simpler implementations http://skunkweb.sourceforge.net/PyDO you simply define a class and supply a bit of extra info to the constructor about what connection/table/fields it represents. Then you can read/write the object attributes and *the in-memory object in turn queries the DB*. Proxy Pattern?

When the object is initially created in memory, an SQL statement grabs the row info and populates the object. Subsequently accessing object attributes DOESN'T invoke an SQL statement - thus reads are cached, writes go straight out to the DB. Cool. You get to use real objects, and the updating of the disk based DB goes on behind the scenes (SQL statements are created & executed for you).

More interestingly....

The way the above OO-DB mapper handles relationships is RM :-) The documentation for PyDO says:

```
Relations and PyDO
------------------
The way you do relations with PyDO is with methods.
For example, if we had a Files class which had an
field OWNER_ID which was a foreign key to the USERS
table, we could write a method for the Users object
like this (a one to many relation):

def getFiles(self):
    return Files.getSome(OWNER_ID = self['OID'])
```

Thus this is exactly the same 'one-line' call to a RM that the RM pattern talks about!! Remember that the big benefit of RM is that you code all your relationship accessing and setting methods (e.g. getOrders, addOrder(order), getParent etc. etc.) as simple one line calls to a RM. Except in this case, each RM request gets converted into a SQL call to a DB on disk, rather than being handled solely by an in-memory RM.

CONCLUSIONS:

I believe therefore that this object to relational mapper PyDO (and possibly others like it) are implementations of the RM pattern, though they also implement the 'object-to-relational mapper' pattern.

These implementations replace a central, in-memory RM with other machinery (some smart base classes etc.) that talks to a real DB on disk/on a server. Thus the work of RM is done by a real, proper relational database :-) and the persistence / mapping problem mostly disappears. :-o

The big benefit of RM is that you must code all your relationship accessing and setting methods as simple one line calls to a RM, which saves you work. An additional benefit (in light of these discoveries) is that by coding all your relationship accessing and setting methods as simple one line calls to a RM, you potentially get automagical and immediate peristence to an external DB.

Thus I'm excited to conclude that in fact it looks like RM *can* scale up to big DB's.

As to whether this architecture can handle client server situations - I'm not so sure. I'm now looking at is how to notify the in-memory objects that their cached info is out of date or that they have been deleted by another user... And I've yet to investigate locking issues...

cheers, Andy

Discussion with James #2

Quoting James Hutton :

James: I'm not convinced that PyDO is an example of RM, because (in that brief example at least) there is no central table of relationships.

Andy: The DB on disk is the central table of relationships.

The PyDO implementation of RM constantly refers (proxy pattern) to the 'relational' DB rather than trying to keep a copy of the relationships in memory.

James: Hmmmmmm, still not convinced. Consider these differences: With PyDo, If I call user.getFiles() twice, I'll get two different sets of file objects, even though they represent the same files. With RM, I would get the same file objects both times.

Andy: Yeah - pyDO creates new objects every time rather than returning existing objects. This needs to be fixed with a slightly smarter implementation of PyDO.

Adding a 'data manager' as described in http://www.python.org/workshops/1997-10/proceedings/shprentz.html would: "After converting a row to an object, a data manager will check its cache for an existing object with the same object ID. If the cache contains a matching object, the cached object is returned instead of the newly retrieved object. If the cache has no matching object, the newly retrieved object is stored in the cache."

I have also written to PyDO's author and asked:

> What springs to mind [as an improvement needed by PyDO] is perhaps a caching system, so that the same objects get returned when appropriate. At the moment two calls to Employees.getSome() will return two lists containing totally different instances. Maybe I could learn to work with this 'feature' and not worry too much about it - but then again, stray objects might get out of date and you never know when you should be doing a .refresh() on them. If only the database could notify each PyDO object when it needs to be refreshed - that would be nice, and might allow multi user use, too. Thoughts?
>
> Read the response from Drew Csillag, the author of PyDO**, who argues that a caching system is not necessarily the answer.**

James: The (A,B) (B,C) => (A,C) thing is easy with PyDo, but hard with RM.

Andy: Right, a smarter RM could theoretically manage it, but I have to admit that I really wouldn't want to go down that pathway - i.e. re-implementing relational technology. But hey - these more complex sorts of relationships are not what you represent on a UML diagram anyway, not directly anyway. UML diagrams just represent one to one, one to many, and many to many - anything smarter is derived/constructed out of from these basic relationship types. RM pattern only seeks to model the minimum, basic relationship types - those that we have to painstakingly wire up every day. So the fact that PyDO (by using a traditional relationship database as it's RM) can do even more sophisticated things is merely a bonus.

DON, THIS NEXT POINT SHOULD INTEREST YOU SINCE YOU HAVE COMPLAINED THAT YOU CANT READILY TAKE ADVANTAGE OF SQL SMARTS IN YOUR PURE OBJECT MODELLING:

Funky SQL

Now when we need to do more complex relational stuff in our OO software e.g. (A,B) (B,C) => (A,C) or getQuarterlySalesFigures(region) or whatever, we usually have to write special methods/classes, data structures, lists and loops that painstakingly pull together this info. Wheras a suitably groovy SQL statement could do the same job quickly and easily.

SOLUTION: With PyDO, which uses a DB as a RM, you can write one line methods that use funky SQL to bring you the info you need, rather than hand coding algorithms etc. all the time. Its the best of both worlds, OO plus the power of proven and mature relational DB technology.

Possible complication to this SOLUTION of using funky SQL as a substitute for lots of OO coding...:

Even if you do get to leverage SQL from your OO program, what sort of objects are returned by the sql call? You have arbitrary number of 'columns' coming back, and there is no object that necessarily maps to the return result tuples/recordsets.

You could invent a new object for each type of returned result, and map manually. Automapping of the PyDO style might be a bit hard, but I need to study PyDO a bit more. This question is really interesting...

Type of Relationship Manager	Benefit	Notes
'Vanilla' RM holding just basic relationship types	Simple one-line relationship methods calling a RM for info. No wiring hassles.	A 'relationship method' is a method on a class that implements or services basic relationship types like one to one, one to many, and many to many e.g. getOrders, addOrder(order), getParent etc.
RM implemented as a proper relational DB (using pyDO style technology)	'funky SQL' Simple one line sql calls to implement our more complex relationship methods. No complex algorithmic hassles in your object's methods. Let SQL do the dirty work.	[A 'complex relationship method' is a method on a class like getQuarterlySalesFigures(region)] Note: see issues to consider to this funky sql idea.

James: RM needs code to transmogrify the relationships list into RDB tables (i.e. there's no DB table called "relationships" so you somehow need to translate the data). That's not an issue with PyDo.

Andy: Right, a traditional in-memory RM needs transmogrification/mapping layer to shovel its info into a DB. But its not an essential feature of RM's that they be divorced from DB's in this way. What is essential to the intent of RM, is that relationships are managed by something smart and central rather than distributed across millions of individual classes, with their individual implementations.

Both an in-memory RM and a PyDO using an external DB as a RM satisfy this intent.

Persisting entities and relationships living in caches

James: Perhaps this will clarify my position:

Consider that example code which has a one-to-many between user and file. If you needed frequent access to a user's files, you'd probably want to cache them, rather than reading them from the DB every time, as the getFiles() function does.

If you were to modify PyDo so that it caches the results of the read, then you end up with the original problem: either you spread the relationships over several objects and deal with backpointers, or you centralize them deal with relationship transmogrification.

So I see RM as something that deals with caches of persistent data, whereas PyDo is something that deals directly with the persistent data and so doesn't need RM.

Andy: (this response not emailed to James, as the conversation had hit a natural stopping point) 22-march-2002 - This last response by James is quite challenging and requires some clear thinking to respond to. Here are some initial responses, which I need to consolidate at some later stage:

Does the introduction of caching really re-introduce the problem of distributing relationships over several objects or in a central place, like a RM? Yes, I think that is right.

Does the centralization of relationships in cache/memory necessarily introduce the problem of transmogrification (having to persist the relationship info to disk) ? I don't think so.

Does the centralization of relationships in cache/memory necessarily introduce the problem of transmogrification?

The centralization of relationships in cache/memory, the way a plain, vanilla RM does, does not introduce the problem of transmogrification. The problem of persistence is there ALWAYS - whether the relationships are spread over the objects, or whether they are centralised in a RM - we still have the job of object to database mapping / persistence

/transmogrification. And in fact, RM makes it *easier* to do your persistence (see original paper, section on persistence). Yes, a PyDO which did caching would also have to transmogrify. Interestingly, a PyDO which doesn't cache talks directly to the DB, and doesn't need to transmogrify.

So I think that transmogrification / persistence / OO-DB impedance mismatch mapping is a different issue to RM. RM is not about caching, it is about centralising relationships. Whether you deal with auto caches, in-memory explicitly maintained caches/objects or directly with a DB on disk, its still RM. PyDO is simply a version of RM that also solves the problem of persistence, since it doesn't cache and thus talks directly to the DB.

Does the introduction of caching really re-introduce the problem of distributing relationships over several objects or in a central place?

I guess that as soon as you make the distinction between an OO object and its persistent state in a DB, then the OO object can be seen as a type of 'cache'. Look at J2EE entity beans - these are in memory copies of the table data entities. In fact the whole idea of programming with objects in memory could be viewed as programming inside a memory cache!!

'Caching X' just means having another representation of X (see Transformation Interface Pattern) and transmogrification will need to occur to translate/map the info from one state to the other. Talking about the "transmogrification of relationship information" and of object information from and to the cache is just another way of saying we have to persist and map objects & their relationships into another format - the database format (in this case).

And if you want to know how each 'cached' OO object relates to each other, then you need to keep this information somewhere too - either in each object, as appropriate, or in a central RM that also is native to that cache / memory.

Is RM merely something that deals with caches of persistent data?

Regarding the statement: "I see RM as something that deals with caches of persistent data" - I would say that RM's principle of centralised relationship management is not *just* an issue that occurs with caches. For example, traditional relational databases are relationship managers. The relationships are spread over a number of tables, but the database as a whole is the RM. Our SQL talks to the central RM, the relational database.

PyDO by talking to a **RM implemented as a DB** doesn't involve a cache, thus RM doesn't just deal with caches.

Ok you can dispute that a RM **implemented as a DB** is a true RM. But it is, because the definition of RM is that relationships are managed by something smart and central rather than distributed across millions of individual classes, with their individual implementations. A DB is smart and central, and manages relationships.

The other signature of a true RM is the fact that your methods that deal with relationships cease to have private and custom implementations of relationship code (e.g. lists of pointers to other objects etc.) but insead have one line calls to a central RM. Both a vanilla RM and a PyDO solution (with the RM as a database on disk) have their relationship methods coded in terms of a central RM. Compare vanilla RM's implementation of a relationship method with PyDO's implementation of a relationship method. E.g.

RM	PyDO version of RM
def AddOrder(self, order): RelMgr.AddRelationship(From=self, To=order) **RelMgr** is the central Relationship Manager we are querying.	def getFiles(self): return Files.getSome(OWNER_ID = self['OID']) **Files.getSome** is a proxy for the database on disk. The appropriate SQL is manufactured. This whole mechanism, including the relational database on disk, is the Relationship Manager.

Regarding the claim that "PyDo is something that deals directly with the persistent data and so doesn't need RM"

Is PyDO is really dealing *directly* with the persistent data? Isn't there actually a DB in the way, mediating all the calls to the peristent data? Of course - that is the relational database, which is acting as a relationship manager. It knows the relationships in the tables - you just ask via SQL and it gives you the answer. You don't care how the tables are stored or implemented - you just stick to the API of SQL.

PyDO is doing what an RM does - using a central relationship manager to store all the relationships and interrogating it whenever it needs to know who is related to who.

Postscript

James: I think Drew's reply makes a lot of sense, but it's a side-issue to my previous email. If you access the user's files a lot, then you can't go calling **getFiles()** every time you want to list them. So you have to cache them. And then you have the issue of backpointers to worry about. So then you need to decide whether you're going to use RM.

Andy: Agreed, as soon as you have cached objects you also need to represent the cached relationships between those objects in some way, be it in a distributed or a central fashion.

Musings on caches, three-tier and persistence layers

25/03/2002

You only need a persistence mapping layer if the way you use your objects is different to the way they are stored.

On the other hand, Object databases (OODB's) present no major conceptual barrier for persistence.

Aren't J2EE's java bean entity beans just objects extracted from the database, given life (by adding behaviour) and surrounded by OO business rules. Changes to the entity beans needs to be persisted back to the DB.

Aren't regular stand-alone apps that have a a well layered design of GUI - object model incl. business logic - and peristence three-tier in spirit? When you take the additional step of locating each layer on different machines/server then you have three tier.

Aren't Web cgi apps that use a SQL server three tier? The Html GUI is one tier, the cgi app of in-memory objects and business logic is the second tier, and the database server (e.g. MySQL) is the third tier.

Running a web server and a sql server on your development machine is three tier.

-Andy

PyDO (Python Data Objects) discussion

This discussion was triggerred by various discussions about Relationship Manager (see above). Some interesting ideas on "persistence layers" and "object to relational mapping" problems were discussed. Interestingly, there seems to be an implementation of an "object to relational database mapper" called PyDO which seems to me to be an implementation of RM pattern, since it implements relationships between objects as simple one line calls to a central RM. In the case of PyDO, the RM happens to be a real relational database, so you get automatic persistence, too.

Here is a reponse from the author of **PyDO**, Drew Csillag when I asked him why PyDO has no caching.

> On a final note, your PyDO has sparked a long conversation between a
> few of my collegues, regarding persistence layers and so forth. I'm
> not sure where PyDO stands in the grand scheme of things - obviously
> there are more sophisticated object to relational mapping
> technologies out there. In this context, what would you say is the
> most needed enhancement to PyDO? What springs to mind is perhaps a
> caching system, so that the same objects get returned when
> appropriate.

IIRC, I believe that there may still be references to something called SDS in the documentation for PyDO. SDS did such caching (actually it cached relations and all sorts of things, it was quite a piece of work, query language, data modelling language and everything -- over top of Oracle).

PyDO was designed to be a realatively thin layer over the database (SDS was quite opaque). The main reason is this: with thick layers, if you need more than the layer will give you, you're basically hosed because going around it (and going straight to the database) may break it in odd ways. Also, if you have DBAs that want to know what you're going to do to their database, thick layers are very difficult to describe and your DBAs will hate you. PyDO was designed with the goal

that it makes the common stuff easy (to do and explain), but advanced queries and things like that (e.g. stored procs), you can go direct to the db (perhaps as a PyDO object method tho..) without fear that something weird will happen -- and still be able to make the users of your data classes be ignorant of what is happening behind the scenes.

In short, I basically found out that while thick ones are pretty and do all cool theoretical stuff:

a. they tend to be opaque and it's really hard to see what they're going to do and explain to your DBAs. BTW: I come from a *LARGE* web shop, and the DBAs, rightfully, want to know if you are going to bodyslam their database.
b. when they don't work, they're very difficult to debug
c. with most of them, you can't just drop them on top of just any schema. They usually have to fit some kind of pattern (e.g. all tables must have an ID column or something) for them to really work.

The thin ones may not do everything under the sun and may have a few ugly bits, but they do most of boring stuff that you'd have to do by hand otherwise and make the hard stuff still doable, while having most of the usability benefits of the thicker ones.

As for features that I wish PyDO had: support for more databases! Given Oracle, PostgreSQL and MySQL (we've got sapdb too) catch most people, but I'd still like to be able to support sybase, Interbase, DB2 and any other databases I can't think of right now and be able to provide a genscript (a script that grovels over the system catalogs/data dictionary/whatever and produces PyDO classes for them -- see pgenscript and ogenscript) for them.

> At the moment two calls to Employees.getSome() will return two lists
> containing totally different instances. Maybe I could learn to work
> with this 'feature' and not worry too much about it - but then
> again, stray objects might get out of date and you never know when
> you should be doing a .refresh() on them. If only the database
> could notify each PyDO object when it needs to be refreshed - that
> would be nice, and might allow multi user use, too. Thoughts?

With caching, this problem only gets worse (as it turns out, it's *much* worse). There's obviously the problem of when to invalidate cache entries, how much object manipulation you're allowed to do before it pushes it to the database. If you want a cheesy, simple relationship cache, you could do something like the following:

```
#in some Groups object
       def getUsers(self):
               if not hasattr(self,'users'):
                     self.users = Users.getSome(GROUP_ID=self['ID'])
           return self.users
```

The other problem that I found out the hard way, is that there is no "generic form of caching" that suits all applications properly. If your caching scheme doesn't fit your application, weird shit happens. My specific case was that SDS's cache was designed with web applications in mind. When used in a newsfeed script (the Reuters feed, IIRC) it got *really* confused (and tended to gobble up memory), mainly because SDS's cache was meant to be flushed fairly often (at the end of a web request) and the feed script was a long running program (well it was supposed to be, but it's confusion didn't allow it to be, we wound up having to flush it's cache every N articles).

Until databases have some mechanism for invalidating entries in some external cache, this problem will continue.

Anyway, this is why PyDO has no built in caching.

Drew.

Drew Csillag

Date:
3/25/02

> Until databases have some mechanism > for invalidating entries in some > external cache, this problem will continue.

I wonder if triggers could be used for this. I've never programmed a trigger, but if you could set one up to call your code every time a change was made (this would be like Observer Pattern), then we could cache to our hearts content, have multi-users and be happy!

Not knowing how triggers work, I'm guessing you can't have a global trigger for any change to a database (too inefficient anyway?) vs. a trigger per table etc. - I'm not sure what is possible along these lines...

-Andy Bulka

Date:
3/25/02

> I wonder if triggers could be used for this. I've > never programmed a trigger, but if you could set one > up to call your code every time a change was made > (this would be like Observer Pattern), then we could > cache to our hearts content, have multi-users and be > happy!

Perhaps triggers could be used for this, where they presumably notify some notification daemon and then the notification daemon notifies individual processes that have references to the appropriate objects (presumably, one would have to do some sort of registration with the notification daemon) to have them (in PyDO parlance .refresh()) refesh their state or commit suicide (if they were deleted).

The main problem here is the infrastructure that's involved. If you have long running transactions and processes (for some definition of long), it makes sense, for short lived processes or transactions, it may make more sense just to get rid of any objects it got than try to do any form of caching.

-Drew