



## Astrum Futura Redux

For the cool folk who have followed the Solar Empire legacy through thick and thin since 1999, those cute .php3 file suffixes, Moriarty's continuing denegration of all other developers, the revolution that open source brought, the sudden push towards OOP, and the...err...security and bug population... Working on anything hitting it's 9th anniversary in PHP is really interesting - 9 times the fun and games 

Astrum Futura was envisaged as a desperately needed update to the long running open source Solar Empire franchise, where players engage each other across a galaxy of 500+ star systems mining resources, building colonies, and generally screaming bloody murder at each other. It's not the most impressive or ambitious of online games, but it's always been tenacious and attracted roaming users who like a quick dash of destruction in their daily diet.

In January 2008, the oft delayed development process will once again creak into action. The current tracer code was built originally using the Zend Framework 0.2-0.6 and famously led that Spring to my long running "Complex Views With The Zend Framework" blog series that gave birth to the Zend\_View Enhanced proposal, and maybe gave Ralph Schindler a few headaches 

. Building anything more complex than a login page was pure heart ache previously.

Now that time is available, the Zend Framework significantly more mature, and we have a lot of legwork carried over from 2006 in the Quantum Game Library (thanks to Jacob Santos), it's about time we got something concrete done. The usual suspects, if interest levels are high enough, can report to the shiny updated phpBB3 forums at <http://forums.astrumfutura.com> - same domain as this blog you're reading.

And we will be applying XP this time - the random running process that usually prevails just doesn't work out well. So get your Selenium gear in place.

Posted by Pádraic Brady in Astrum Futura, PHP Game Development, PHP General, PHP Security at 17:01

## Quantum\_Db: The QGL Data Access Object Implementation

Back in 2005, the Quantum Star SE project caught the Object Relational Mapping (ORM) bug and I took a first stab at implementing something that was simple, fast and easy to use. Until this stage I was a regular user of the Data Mapper pattern (among others) but since switching to PHP and developing open source projects found it to be largely unnecessary. The root problem is akin to hammering nails with a sledgehammer - too much complexity being implemented for a simple underwhelming task.

Here in 2007, with the QGL in development I found myself re-assessing my previous experiences and finding it attractive to develop something less complex and hopefully faster than a full scale ORM solution. Taking some previous work I did, I opened a branch in the QGL subversion repository and set to work. The results are slowly coming to fruition - the DAO, Row and part of the Driver families are shaping up going by the unit tests.

So what is Quantum\_Db (in practice)? It will be a family of classes which form a Data Access Object (DAO) Framework when complete. This post is largely a ramble about current progress and intentions - so if looking for a complete solution you're in the wrong place as the current code is strictly in-progress. For reference though, the source code is under the New BSD License. It's incomplete, but available from its subversion branch at:

[http://quantumstar.svn.sourceforge.net/viewvc/quantumstar/branches/Quantum\\_Db/](http://quantumstar.svn.sourceforge.net/viewvc/quantumstar/branches/Quantum_Db/) and the forums can be found for the QGL in general at <http://forums.astrumfutura.com/viewforum.php?f=25>.

In this specific implementation of Quantum\_Db, the DAO class (Quantum\_Db\_Access) is a Singleton which attempts to offer relatively basic CRUD methods capable of use on any valid database table record. A large part of its responsibility is generating CRUD type SQL. So you could have a User record, which could be saved by passing it to a DAO save() method. It's not quite as clear cut, since to reduce the mass of code a DAO implementation often needs the "record" is encapsulated in a container called Quantum\_Db\_Row. This Row class can represent a single table record, but also doubles as a data container for setting up SQL conditions (think of WHERE, IN and INSERT clauses).

One of the goals intentionally targets simplicity, so there is no Propel-style Criteria object to mess about with. There will be in the future, but for simple needs, a simple DAO framework is sufficient as an initial start. A simplistic use case is adding a new User record. Some code showing this task:

```
require_once 'Models/User.php';
// Fields are public properties handled by __get/__set in the
// Quantum_Db_Row class. The User class is actually a simple
// subclass defining valid fields and primary key(s).
//
// We do not specify an Id value for new Users. The DAO class will
// assign this assuming Id is an auto-incrementing value for MySQL
// or a Sequence for PostgreSQL.
$user = new User;
$user->name = 'Pádraic';
$user->email = 'padraic.brady@example.com';
$user->password = sha1('password');
$user->country = 'ie';
// Call on DAO to save this record to the database. Obviously any
// database errors will result in Exceptions thrown by the DAO so
// the database table rules (primary key, unique key, etc.) all apply.
$user->save();
```

The first thing you can note is that the DAO class (called Quantum\_Db\_Access is not directly called. Instead all Row objects carry a reference to the Quantum\_Db\_Access singleton so Quantum\_Db\_Row::save() proxies to Quantum\_Db\_Access::save() - we only need one generic DAO object. If one desperately needed the DAO they could just place a call to Quantum\_Db\_Access::getInstance().

```
$dao = Quantum_Db_Access::getInstance();  
$dao->save($user);
```

In fact, you can equally access the underlying database extension driver to run queries directly using the following. Of course Singletons have their own evil uses, so ideally you'd pass around the object reference instead of littering source code with static getInstance() calls which explicitly refer to the class name.

```
// Example of accessing the driver to drop the User table  
$driver = Quantum_Db_Access::getInstance()->getDriver()->exec("DROP TABLE user");
```

Retrieving records is more interesting. It's relatively simple to extract a record based on a primary key:


```
$user = new User;  
$user->getByPk(1);  
var_dump($user->asArray()); // echo array of User (Id=1) data
```

It's much more complex to fetch rows of a very specific nature. Even worse, without a Criteria object using the MySQL IN clause is a challenge. However for those where the SQL values depend solely on simple field values a neat solution is "named queries". A Named Query (check the excellent introduction at: <http://www.tonybibbs.com/article.php/PropelDAO> regarding something similar for Propel) is an SQL string containing value placeholders (e.g. for use in a PDO::prepare()) stored in some central file. This has a few benefits and disadvantages.


The main benefit is the centralisation - like OOP, centralising logic (in this case non-generated SQL strings) reduces duplication and encourages reuse. Allowing the named queries to be indexed (for example, within an XML hierarchy) by a name and model allows them to be easily located and prepared by the Data Access Object. The disadvantages are of course that it's still limited. For example, use of a MySQL "SELECT id, name, email FROM user\_table WHERE id IN(1,2,3,4)" isn't likely a storable named query since the IN() range of id values is too variable. But for many queries it's a good fit in the absence of some form of Criteria class.

Enough of the DAO class however. After beating it to death, there are still pieces to be completed when it comes to allowing complex SQL clauses. The main point is that for simple use cases it is a good match - and simple use cases are extremely common in many of my open source endeavors.

To backup the Data Access Class Quantum\_Db also implements a Driver and Results class family. Again simplicity is key, so it's being written under the presumption that PHP5's PDO extension is available. The Quantum\_Db\_Driver\_Interface file actually mirrors the PDO interface for this reason. Of course we can't forget mysqli (the most popular PHP5 database extension) or pgsql and the rest. Quantum\_Db\_Driver defines an interface to enable additional extension type support. Personally I only intend adding a light abstraction over ADOdb-Lite - it's not worth duplicating Mark's already existing library.

The Results class, the one part no code exists for yet - don't be worried I develop using TDD and unit testing so it's perfectly normal 

. Quantum\_Db\_Results will aggregate the results of queries into a class implementing PHP5's SPL Iterator. To be specific, the Iterator methods will be tied to each underlying Driver's fetch\* methods. This should allow continued use of Lazy Loading during iterations, and allow continued use of foreach() etc.

Here ends my ramble for now. For those working on the QGL or similar code maybe it will provide a few of my thoughts on the topic - assuming you can summarise the rambling of course 

Posted by Pádraic Brady in Astrum Futura, PHP Game Development, PHP General at 15:25

Friday, February 2, 2007

## QGL under the New BSD License

The Quantum Game Library ("QGL") is a PHP5 library aimed at creating a bunch of reusable classes for use in PHP games. We're still in early days but with two dedicated developers, lots of feedback and suggestions we've already put together a few starting solid components and have others in progress.


Over the last few months the QGL has evolved from a minor offshoot of the Astrum Futura project into a fully independent library. I guess everyone on the team saw a generic game library as something really useful; we're all game developers on varying projects.

To reflect the library status, as well as to widen the net for potential users and developers we have decided to move from the GNU General Public License (which rules Astrum Futura) to the far more liberal [New BSD License](#). We've also provisionally determined that copyrights should be centralised to a degree, this currently is not in operation since the number of contributors remains manageable but may be (pending contributor feedback obviously) in the future.

Given licensing is now decided, file header updates in progress (if gradual), and some useful components are hitting stable, it's likely we'll make a public preview release in the near future. Given our focus on doing things right, this will only occur once documentation is completed and our unit tests verified for good measure.

As a summary (feel free to request details on the [QGL Forum Section](#)) the current near-stable component list includes:

- Quantum\_Db
- Quantum\_Coordinate
- Ai\_Pathfinding
- Quantum\_Map\_Measure
- Quantum\_Map\_Grid
- Quantum\_Turing

This is far from complete, and there are literally dozens of possible additions. Those in planning are already shaping up to offer some cool second iteration functionality. That's not even including the planned ext/qgl PHP extension... 

Posted by Pádraic Brady in Astrum Futura, PHP Game Development at 11:47

Tuesday, January 30, 2007


## Factions: Social Control on Newbie Bashing

A recent discussion was sparked on the [Astrum Futura forums](#) regarding player retention and new player protection. It was one of those times when the discussion branched to a higher view - a wider look at how players interact on a large scale.

The current suggestions called for players to be penalised for undesirable actions, e.g. attacking a new player who is incapable of defending him/her self. This remains a long standing issue in many online games. AF could resort to enforced levelling - limited who a player can attack, and be attacked by. But this is all artificial - in the real world it's open season on anyone.

Cyberlot (Richard) raised the question of penalising through a Karma rating. As a player attacked newcomers they would attract negative Karma marking them out to other players. I can see the rank of "Newbie Basher" emerging as an undesirable outcome for many players. But Karma while adding some measure, does not address enforcement. Exactly who enforces what and when? Who sets the standard for interacting with other players?

From there we hit the subject of Factions. A Faction would be a specific group of players sharing a common interest. This obviously includes Races, but is also extendable to a Merchant Guild or a Pirate Clan. The point is to give players an instant group they are a member of. Now where it gets interesting in how the player and game logic would respond in the presence of Factions.

Say you have a Faction, Earth Humans United (go Terrans!). What happens if a member of this Faction attacks a new player who is joining the Earth Humans United Faction? Can't have allies killing each other off (even if it might reflect reality 

). The answer is in the Karma. Killing a new member of your own faction is stupid, damaged the Faction, and makes other players less likely to join it. That member should sur be paying tax to the Faction, which is needed. Such a player would immediately get negative Karma.

So we have Karma, and we have Factions - how do they relate?

Karma is a measure of one's standing within a Faction. If your Karma falls below a minimal level you'll be declared an enemy of the Faction, and then its time to start running before you're hunted down by your peers. Where do you flee? If you're lucky, some other Faction still has you as Neutral.

But this is where Newbie Protection get's interesting. It's not Faction specific - your bad Karma is classified when killing a new player as a "Stigma". It decreases Karma for **ALL** Factions. It's the ultimate punishment - imagine playing when nearly all the Factions refuse to grant you trade access, when your planets are penalised with "special tax charges", when other players are unable to maintain an Alliance with you once you become Factionless (Outcast). In short, if you gain a reputation for killing new players, you'll be kicked, booted and sent to Hell with a one way express ticket. No Admin interference or funny AI fleets required. Society will put you straight or else...

More on this later as details are worked out, but it looks like it's a feasible and supported suggestion.

Posted by Pádraic Brady in Astrum Futura, PHP Game Development, Quantum Star SE at 17:37

## Thoughts on a Unit Testing and Test-Driven Design Experience

Creeping closer to Spring; it's 1 February if you follow my calendar. One of the things which stand out in my online activities as it touches on PHP is the 3 month period spent working on the QGL (Quantum Game Library). The library is just reaching its 200th commit, and should pass that barrier later this evening when I get around to a small class naming change.

Sticking with the QGL, it's a small side project which sprung up from a vague idea. Those of us working on the related Astrum Futura game project, knew a standard game library of some description would pay dividends for future projects by centralising a general set of useful classes. It was shortly after this concept was solidifying when Jacob Santos joined the team with his imaginative vision of implementing AI algorithms.

The one thing that really stands out from the usual project hustle and flow, was our shared wish to use Test Driven Design and Unit Testing. It's very difficult to use it on anything more than personal projects at work where developer resistance (though falling) is still an obstacle. So the QGL was an all too rare opportunity to cooperate in a team led effort focused around the test-first approach.

So how did we fare?

Personally, I think a lot of what we accomplished to date (bearing in mind the time limitations) was a result of TDD, and of course Unit Testing's benefits in general. There were times where we were working on complementary components such as Jacob on `Ai_Pathfinding`, and myself on `Quantum_Map_Measure` where TDD led to simpler more manageable classes. Of course the unit tests supported our efforts with style. Throughout the process we were both refactoring code, and tweaking interfaces. Without the unit tests we would have reached deep treacherous waters as our respective efforts slowly drifted apart.

I also think poor Jacob may have become test infected. He seems to be recovering, so I'll have to wind him up later over the current two failing tests he introduced! I haven't heard any rumours of a bald Jacob Santos so presumably that odd side symptom has yet to rear its ugly head.

Back on track... The most visible benefits of the TDD + UT approach during the last three months can be summarised as:

- Easier Refactoring (perform a refactor, check tests still pass, rinse and repeat)
- Immediate Feedback when code goes wrong (big fat red bars in the testing results. Hard to ignore!)
- Enforcement of Standards (tests discourage questionable hacks/shortcuts)
- Regression Testing (if something goes wrong, we add test(s) to prevent repeats)
- Force Feedback (not just a console controller type, failed unit tests spark attention from other developers)
- Code To An Interface (TDD and unit testing teach you why if you're a quick study)
- Testable Code is Better Code (adherence to practices which generate clean, focused, flexible classes)

I'm sure there are others, but these were the most obvious over three months of development across 200 commits to subversion from two developers. If the immediate benefits are not a motivation to become test-infected the future benefits might. The one thing I know from past experience is that unit tested code tends to be easier to maintain and generates fewer bug reports.

Now if only someone miraculously removed the initial pain a developer experiences when starting to learn TDD and Unit Testing (the pain that explains why many quit before the summit of the learning curve is reached) we could get more folk on the TDD bandwagon. One unfortunate fallout from following a test-first approach is that it necessitates any

new developers becoming educated. No easy task when dealing with a vast population of wannabe programmers taking their first few running jumps with PHP by showing interest in open source projects.

Posted by Pádraic Brady in Astrum Futura, PHP Game Development, PHP General at 17:58

Tuesday, November 28. 2006


## Astrum Futura: Performance and Optimisation

I thought, as reference, I'd do up a post concerning performance and optimisation in Astrum Futura. Mainly to get my thought processes in text, and take a look at where performance is lacking.


The main driver of performance at the moment, in the absence of reams of game specific code is the Zend Framework. The ZF is currently reaching a v0.60 preview release in subversion (should be release next month) and an interesting discussion took place last week on the ZF mailing lists concerning [this blog post](#) by Paul M. Jones comparing four framework. In a simple minimalistic "Hello World!" application using each framework's most minimum setup possible, and measuring requests per second (on a local PC), the ZF came last.

The main causes of this was the pure OOP approach. Each component has a few Exception classes, each in a separate files, which were loaded (as all ZF files are) using `require_once()`. Add to that the possible use of a static loading method which did weird and wonderful I/O filesystem checks and all that work created a major drag. That may be addressed properly in the future - until a 1.0 release the idea of optimisation is still largely considered premature optimisation. At least one small improvement was made - the use of Reflection has been largely removed from the many Controller classes. Reflection in PHP5 is nice, neat, and very expensive.

The Astrum Futura approach is to wait and see. Until then we've added an `__autoload` function, and are prepared to strip all instances of `require_once()` calls from files we distribute in a release. That in combination with autoloading would ease the burden a fair bit. A release is months away of course, so wait and see is the only reasonable stance right now.

Another future performance driver is the database. This gets interesting however, in that we can't consider the database purely in isolation. By itself we can rely on a mix of query caching (on the few shared hosts who actually bother to check their MySQL configuration), the use of InnoDB and MEMORY (HEAP) storage solutions for tables (again MySQL), and the usual pretension that 4th level normalisation doesn't exist 

In AF, we also have to deal with how database results are handled. In general terms, an AJAX frontend does not typically rely on HTML input. Rather we receive responses in a strict data-only format such as XML or JSON. JSON in particular is the most useful since it's Javascript's native object notation. XML is more complex since it requires using Javascript's DOM to manipulate. One of the focuses for near static data will be utilising caching extensively - not just server side, but also in the client. One of our main concerns, and one of the reasons we're coding to PHP5.2, using MySQL more productively (no MyISAM crap here), and actively seeking ways of caching data, is the fact that the interface is AJAX enabled.

AJAX gives a nice warm fuzzy feeling when you see it in action. But if used extensively in the wrong way it can create a ton of problems. The simplest explanation is that AJAX should nearly always increase a user's productivity - it's not just eye candy 

. The more productive they are, the fewer server requests they should need for a given task. The disadvantage is that you can easily wind up with a task which rather than taking one or two big page reloads to complete, takes a dozen or more small AJAX updates. If those small updates are slow, and add up to a larger request time and larger bandwidth takeup than the single big request you started with - then something went a bit wrong. Caching both on the server side, and especially on the client side reduces the number of small requests for static data. The fewer the better...

Now, if only these good intentions converted into reality in the near future...



Posted by Pádraic Brady in Astrum Futura at 00:47