

Re: Poly Couples

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2006-07/msg00425.html>

- *From:* "topmind" <topmind@xxxxxxxxxxxxxxxxxxxx>
 - *Date:* 14 Jul 2006 13:29:00 -0700
-

Sasa wrote:

topmind wrote:

Sasa wrote:

topmind wrote:

Fine. Oop is great for device drivers (like print drivers above), animals, and shapes. But can we pleaaaaaaase get something that reflects what real people see in the real workplace? The vast majority of

Um, drivers are pretty real life example, just as any other "non custom business software" example mentioned in this thread (sorry for nitpicking).

In my fairly limited experience, we had following stuff:

1) Parallel versions of essentially same application. There were medium to large variations in look&feel (but not in the data being edited on the forms), the business part was the same, but some features were enabled in some versions.

Why did you have parallel versions?

Re: Poly Couples

Because it was requested from us :-)

Essentially – different versions for different clients. For example, there was full blown version with relatively classic UI and there was stripped down version with fancy looking UI. I think total number of versions was >10.

Couldn't you use regular functions for that? Just swap routines such as DrawScreenX(...) for each customer. Maybe keep the presentation functions in a sub–folder, and don't overwrite that subfolder when updating the versions. (Of course there will probably need to be some minor tweaking because new requirements may need new interfaces.) Polymorphism would only tend to make more sense if the *same* EXE used multiple different formatting variations, such as dynamic user preferences. (And even this can be driven with functions, depending on the language.)

3) The data is highly hierarchical, as presented in the UI. Logically, some actions on the higher level can affect some or all parts of the subtree.

What data is hierarchical? I would like to inspect the hierarchy.

Maybe I wasn't clear enough. The requirements wanted that the data is presented as the hierarchy. Consider something like windows explorer – tree on the left, input dialogs on the right. If you are on the level 2 and edit some stuff, some subset of the subhierarchy could change (entire tree items could appear/disappear, some subcontrols on one dialog in subhierarchy could be disabled or disappear)

You don't actually use inheritance for each level of the GUI tree, do you?

4) Some simple DSL exists to enable non programmers, domain specialists, to manage validation rules, dynamics of the hierachy (when should some elements appear/disappear depending on the user input).

5) There are ca. 700 input dialogs. Some of those appear

Re: Poly Couples

more than once
in different contexts (same form, for editing different data).
Some are
dynamic in the sense that they can appear zero, once or any
number of
times, depending on user input.

Why couldn't p/r do the same thing?

What's a p/r?

Procedural/relational

and I feel
much less paranoid when editing the code. It is also easier for
me to
analyze the code. We worked in fairly high paralelised mode,
due to
strict interfaces and separation of concerns, we managed to
work
independently and integrate without significant problems.

Schemas can do something similar. Each section only worries about what
it puts or reads onto the DB and not other code modules (other than
shared libraries).

I'm willing to give the benefit of the doubt that OO is not
optimal, but
I still fail to see the arguments coming from the hard OO
critics.

Other than that the case for polymorphism (which to me
means programming
to interface and varying implementations) would be unit
testing – you
want to test a module and want the test to fail only if there is
error
in that module and not in any other on which it depends.

I would like to inspect such code.

Re: Poly Couples

I'm no unit testing expert. Keeping that in mind, consider following example:

I1 <- TestableModule -> I2

Since testable module depends on interfaces, by providing simple test implementations, one can test the module in the isolation. Tests will fail only if there is an error in TestableModule or in the test itself. The implementations used in production are not included in the test.

Modules are not an OO-only concept.

As for swapping databases, I already mentioned in some other thread the contrived example – imagine writing the application for one bank, now imagine you want to sell it to some other bank. The business process is essentially the same, the existing UI is perfectly fine. However, they insist that you use their existing database which is from a different vendor, and the structure is different than the one you use originally.

Okay, but that is a known up-front need. Plus, it can be done with

Not always, and it can be secondary. Consider that you deal with single client. So you start with your application, code it, test it and deploy it. And then the company tries to sell application to other potential clients. And one client says "This is great, but we require that you use our existing database".

I encounter a lot of What-If scenarios as I look at designs, not just wrapping vendors. One has to make a decision to spend extra code to prepare for such, or skip it. Software design is very similar to investment decision making.

My general point is that you usually don't know where the change will be. By breaking the system into small, independent, cohesive components

Re: Poly Couples

(which to me means decoupling), one has the system which is easier to change.

Not necessarily. Things you separate may start growing or changing together such that a formal interface between them is a red-tape pain. Indirection and black-boxing is not a free lunch.

Sure, it will not cover all possible change scenarios, but IMHO it still helps a lot.

The finer the granule of the module division, the better the chance that modules will get reused – both in the changed version of the same system, as well as in some other system.

Dealing with future changes is not the only motivator. IMHO it is easier to develop and work with small, independent modules, they are easier to develop, integrate, analyze, debug and test.

Agreed. That is why I split designs up into "tasks".

functions. Generally you would target about 4 vendors (SQL-Server, Oracle, DB-2, Postgre), and thus infrequently have to add new CASE items. You can't target every DB vendor because then you would have to test all jillion vendors. And if you did have that many, why not put the SQL in the database and then dispatch it that way? Special characters would mark variable insertion place-holders in the SQL.

table: SQLstatements

queryID
DBvendorID
descript
SQLtext
parameterDescript // (perhaps use a param table for this)

table: DBvendors

venderID
venderDescript // example: Oracle, DB2, SQL-Server
useAltID // vendor to use if statement not available for given vendor

This way you could make some nice screens to bring up, report on, and inspect all the SQL statements for the different vendors. I've seen similar tools to allow DBA's to change SQL without having to fiddle with code, although it had nothing to do with vendors.

Re: Poly Couples

One can add a new DB vendor without compiling a single line of new code.

What if structure of the database varies, and you have business calculations (which is common for all systems) in SQL?

To put it more systematically – consider following constraints:

1. You ship the application to multiple clients.
2. The databases systems (both vendors, and the structure of database) are different. The clients don't allow you to use your own database, or to change the structure of the existing one, which you must use.

Most of the time clients are not that picky for biz software. Databases are common and expected. In some cases one can use a compiled-in one so that the client does not know the difference. In the worse case, convert the table into a giant CASE statement. One does not have to maintain the changes in the case list, though. Generating the new case list from the internal DB is then just part of the build process.

3. The business logic is mostly common. Some subtle variations can exist.

Use the same structure or very similar structure to only store parts of an SQL statement instead of all of a statement. Remember, it has a "default" option that kind of serves the same purpose as inheritance. Thus, one does not necessarily have to code each peice if they are the same for a given implementation.

How's that for abstraction? The mother of all DB-vendor-wrappers is a DB! Out-poly that!

You might be right, wrong, or it comes down to cultural differences.

I somehow don't fancy the idea of having a lot of code snippets stored in the database. I prefer OO with statically typed language for the code (but of course, it depends on the nature of the problem). With appropriate IDE, I can work easily with the code, and have features such as refactoring, debugging and compile time checking. It doesn't resolve all possible problems, but it helps dealing with some of them.

I prefer dynamic or type-free languages. It makes the code simpler and smaller, faster to test, easier to make into declarative when needed,

Re: Poly Couples

and more easily meta-tizable in my opinion. Heavy type programmers and "dynamic" preferring programmers seem to have endless fights over which is better. I say it may just be a subjective preference. Some think better in "types", some don't.

Sasa

-T-

.