

Re: What doesn't lend itself to OO?

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2004-08/0809.html>

From: Mark Nicholls (*Nicholls.Mark_at_mtvne.com*)

Date: 07/29/04

Date: 29 Jul 2004 05:22:23 -0700

> >>>> *Just out of curiosity, though, why shouldn't the clock service be*
> >>>> *stateful? It isn't really a resource issue because there only needs to*
> >>>> *be one instance.*
> >>>
> >>>
> >>> *I think we need to be clear about what 'stateless' programming means*
> >>> *in this context, I don't particularly think it means state rather than*
> >>> *specific identity.*
> >>>
> >>> *If the clock service has identity then the client looks like...*
> >>>
> >>> *// create a specific instance of the clock service on the server*
> >>> *CGMTClockService clock = new CGMTClockService();*
> >>> *// invoke GetTime on that specific object*
> >>> *Console.WriteLine(clock.GetTime());*
> >>
> >> *The first line exists in the server (presumably in the setup code).*
> >> *Where does the second line live?*
> >
> >
> > *My psuedo code is possibly the MS view (COM or .net) view of remoting*
> > *objects between client and server i.e. as far as the client code is*
> > *concerned it is unaware that "new CGMTClockService();" creates a local*
> > *proxy and instructs the server to constuct the real object.*
>
> *So you're saying /both/ lines are in the client and all of the server*
> *communication is hidden in the implementation of CGMTClockService,*
> *including network boilerplate?!? If so, doesn't that make*
> *CGMTClockService kind of a god object since it must understand the*
> *semantics of clocks, network protocols, proxy strategies, and server*
> *interfaces?]*
>
> *[OTOH, pursuant to the COM discussion at the end, it is might well be*
> *consistent with that sort of composition. B-)]*

I don't see it as abnormal (but then maybe I have been indoctrinated),
depending on the configuration data associated with CGMTClockService,

comp.object: Re: What doesn't lend itself to OO?

effectively a factory will either create it locally or remotely and in a puff of OO magic insert a proxy stub between. The remote object is identical to a locally created one, but has a stub talking to it rather than client code.

>
> >
> >
> >> *If, as I suspect, it lives in the client, then I don't like*
> >> *"clock.GetTime". The client should be making a generic request for the*
> >> *time from the server with no knowledge that the server implements that*
> >> *request by instantiating a clock object.*
> >>
> >
> >
> > *It was an example of a 'statefull' service i.e. one with a client*
> > *specific identity, and why it was bad.*
>
> *Sorry, but I am thoroughly confused. If 'clock' is instantiated in the*
> *client, I don't see a problem, other than cohesion, because everything*
> *is hidden in its implementation. The server can do whatever it wants*
> *about providing the time. If 'clock' is instantiated in the server, I*
> *don't see why the client should know that it exists; the client should*
> *be invoking a GetTime server interface method so that the 'clock' server*
> *implementation is completely hidden.*

A clock proxy is created in the client and a clock object is created in the server. The proxy 'knows' where this *specific* object is, but how do you manage the lifetime of the object on the server? if the client dies, the object potentially lives forever...so we get into pinging and timeouts and leases and all the other messy stuff.

There is something in .net that allows programmers to still program in this manner AND not have the direct 1:1 correspondence between proxy and remote object, where an object is created and destroyed on the server for every method invocation BUT obviously that object can have no state.

>
> *I suspect I have a more fundamental problem in that I don't understand*
> *how you are relating identity to statefulness. So I think you need to*
> *put some more words around that as well. (I have my own comments on*
> *identity vs. statefulness below.)*

An object may or may not have a set of properties or state, lets assume in general it does.

An object has a type which defines a set of operations/behaviours that an object must conform.

hmmmmmm, I'll ramble a bit...

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

I order to produce behaviour I need to be able to allow the behaviour to interact with the state, so we have an object, but to allow me to have different combinations of state and behaviour I need multiple objects and so to invoke specific behaviour from this set I need to identify a specific object, and this is identity.

If the set of state is a single member and I only have 1 type and no subtypes I can use a singleton.

If there is no state but there are multiple subtypes then I *do* need identity to identify objects of specific behaviour and subtype – even though there is no state.

The point about 'stateless' programming to me is in fact identityless programming (or programming against a logical singleton).

i.e.

if you have diverse state => you require identity

if you have diverse behaviour (subtypes) => you require identity

>
> >>> *Now what happens on the server if the client crashes after the 1st
> >>> line, there is an object with no client, and we need to create all
> >>> sorts of dreadful strategies for making sure it is still needed –
> >>> timeouts, pinging etc.*
> >>>
> >>> *What happens if I want to create a farm of two servers if clock is
> >>> created on server A, the load balancer needs to know this else it may
> >>> send the request to server B and this specific clock doesn't exist
> >>> there.*
> >>>
> >>> *I agree, we have different views of statelessness. B-) I see this as
> >>> an orthogonal problem related to registration, protocols, and
> >>> implementing a bunch of interoperability requirements.*
> >>>
> >>>
> > *I agree they are mildly different but state => identity, if there is
> > only 1 set of state then we need only 1 identity i.e. a singleton i.e.
> > the subsystem itself, but the principle problems to me are associated
> > with the servers supplying multiple objects (identities) to the
> > outside world, and the overhead associated with maintaining these
> > objects. Not orthogonal but a subset of the same issue.*
>
> *But we are talking about different subsystems, each abstracting the
> problem space in its own way. In most server contexts one abstracts
> different <usually conceptual> entities. So the identity of entities
> the client abstracts may not be semantically meaningful on the server;
> it is just another element of a data packet that the server entity
> processes.*

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

- >
- > *In the clock case the server instantiates a single clock object*
- > *regardless of how many clients need the time service. The clients have*
- > *no need to instantiate a clock at all, except possibly as a surrogate*
- > *for the server interface at the OOA level. The clients simply invoke the*
- > *service.*

You are describing an implementation, I was comparing 2. The server could instantiate a new clock per client – there are good reasons to do this e.g. contention on resource – it is possible to do this behind a stateless service layer BUT this is not how off the shelf middleware has traditionally worked.

e.g.

RDB access, the client accesses a specific connection object on the server via a local proxy. There is indeed 1 database entity, but there are multiple diverse connection objects and multiple diverse state associated with each connection, if a client dies the database server has to work out how to identify the dangling object and what to do with after that.

You may well be describing how you work, but you may be more enlightened than mainstream middleware designers.

- >
- > *In the case of persisting client objects, the object identity is*
- > *embedded in the message data packet. The server only needs one object*
- > *to convert any data packet into a SQL string or a native DB engine*
- > *request. In that case the entity identity that is important to the*
- > *client has no semantic value at all to the server; all the server needs*
- > *to know is that the value in position N in the data packet goes into*
- > *clause M of the SQL statement.*

Yes I agree this is a way of creating a stateless service layer by persisting the state to an RDB and allowing the client to reference it via a handle or lookup.

- >
- > *I agree that identity is /indirectly/ an issue because if one persists*
- > *the message data packet data in the server between requests, one will*
- > *have to have some sort of identity for those persisted data packets. It*
- > *will be convenient to use the embedded identity, but the server can*
- > *provide its own. While the server identity will map 1:1 with the client*
- > *identity due to the RDM, it is semantically different because the client*
- > *will be instantiating Customers and Accounts while the server will be*
- > *instantiating Messages or DataPackets.*

And that data and its lifetime needs to be managed.

Take sessions on the web, these can be persisted to an RDB as you describe, and the identity and state of the session will exist in the

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

RDB. How do you manage the lifetime of this session? does it last forever, should you simply let crashing clients leave there orphan sessions hanging around? Possibly yes, it may well be more trouble than its worth to clear them up, but its not ideal BUT if the session exists in the client and client crashes there is no state for the server to worry about and the server is truely stateless.

I'm not suggesting this is the way to write web applications just the overhead involved in storing anywhere else except at the client.

>
> *That's a pretty indirect dependence so I think it is more useful to*
> *think in terms of the core issue: persisting state data in the server*
> *between requests. Identity is just one special case of state data.*
>

special being the operative word, because without it you cannot access any state or any defined behaviour.

The set of problems associated with 'normal' state is thus a subset of the set of problems associated with identity.

>>
>>
>>>*I see stateless objects as a mechanism to deal with limited resources.*
>>>*If objects have state variables, each instance needs to be created to*
>>>*persist those variables. If there are a lot of them (e.g., many*
>>>*shopping carts active at a POS site or, worse, many cached web pages),*
>>>*one will run out of memory and begin page faulting and the DB is enough*
>>>*of a bottleneck. One also has the problem of dealing with discarded*
>>>*objects (i.e., the client went away leaving stuff behind), otherwise*
>>>*physical disk space becomes a limitation.*
>>
>>
>> *I agree but*
>>
>> *CGMTClockService and CCETClockService are both stateless but I still*
>> *wouldn't implement them as individually instantiatable (by the client)*
>> *objects but either as 1 stateless service or (debtably) 2 stateless*
>> *services – I don't have to though, and vanilla OO would not do this,*
>> *it would do as I have done above and did so for many moons.*
>
> *You've lost me again. Where does CCETClockService come from? I don't*
> *see it in your example above.*

as you know....

CET = central european time.

GMT = greenwhich mean time

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

so CCETClockService provides slightly different behaviour to CGMTClockService BUT they are both stateless but the fact they are of two different types means I need identity to identify each.

>
> >>>>> *If possible on the client BUT I accept there are a whole raft of*
> >>>>> *scenarios where this is unreasonable and unwanted i.e. the raison*
> >>>>> *d'etre (!) for the service is to supply some sort of persistence.*
> >>>>>
> >>>>> *There's no free lunch; wherever state is maintained persistently there*
> >>>>> *is a downside. If it is in the server, then one has resource issues in*
> >>>>> *a multi-client environment. If it is in the DB, then one has the*
> >>>>> *performance hit of additional physical I/O access and one increases the*
> >>>>> *likelihood of needing deadlock resolution. If it is in the client, then*
> >>>>> *the client needs to know a lot about the persistence limitations and*
> >>>>> *there will be encoding/decoding overhead into generic formats like XML.*
> >>>>> *(Not to mention that most of the client state will originally come*
>
> >>>>> *from the DB through the server anyway; all one is really doing is*
>
> >>>>> *optimizing the access.)*
> >>>>>
> >>>>>
> >>>>>
> >>>>> *If no persistence is required, then there would seem to be no need for*
> >>>>> *the extra headache. Keep the state on the client and make the*
> >>>>> *interface a completely stateless one, all state needs to be passed in*
> >>>>> *to get a given result.*
> >>>
> >>> *You've only removed the server headache. In doing so you have given the*
> >>> *client a brand new set of headaches. Client requests that were*
> >>> *conceptually separate before because of inherent sequencing in the*
> >>> *problem solution are no longer independent and each request may have to*
> >>> *be aware of others. Basically we have:*
> >>>
> >>> *Request A provides state data {S1, S2, S3}*
> >>>
> >>> *Request B provides state data {S4} but there server needs {S2, S3} as*
> >>> *well to process {S4}.*
> >>>
> >>> *If B always follows A in the solution, there no reason for whoever*
> >>> *generates B to worry about any data other than {S4} -- providing the*
> >>> *server persists {S2, S3}. Otherwise, whoever generates B must be aware*
> >>> *that the {S2, S3} data from A is needed and include it.*
> >
> >
> > ???
> > *If the server holds absolutely no state then the client simply orders*
> > *his requests in the correct sequence to provide the correct behaviour*
> > ???
>

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

- > *However, the client now has to coordinate across requests to collect all*
- > *the data needed for each request _even though that data was supplied*
- > *with a previous request and has no relevance to the context of the*
- > *current request_. That may be quite counterintuitive to the normal flow*
- > *of control of the problem solution.*
- >
- > *Remember messages are supposed to be announcements of something the*
- > *sender has done*

That is your take and not the norm in mainstream OOP, to me messages can be events or commands, and in my tools commands predominate events,

- > *which is usually a state change. Such announcements*
- > *would logically only include the data relevant to that state change.*
- > *However, if the server does not persist data between requests, the*
- > *announcement effectively must include the state changes from previous*
- > *requests. IOW, the context for sending the request in hand must also*
- > *understand the contexts of other requests made previously.*
- >

My mind rightly or wrongly is looking at the from the command perspective and it's difficult for me to see it from an event perspective, in fact my other post is really about this issue.

- > *Since one is only dealing with data, that is not usually a big deal*
- > *because in practice it is likely to be specifically defined in the*
- > *server interface for the request. So the client just has to rummage*
- > *around to encode the data in a data packet. However, it can have some*
- > *subtle pitfalls. For example, the client must ensure that the data is*
- > *still there to collect in later requests and that is hasn't been*
- > *modified in a manner that would affect the server's view of consistency.*
- > *IOW, there is a potential for some rare but nasty data integrity problems.*
- >
- > *[Not to mention performance and bandwidth problems. The client has to*
- > *encode more data per request and the requests are larger. I've heard it*
- > *argued this is actually good because it distributes processing. What*
- > *that argument ignores is that the processing wouldn't be done at all if*
- > *the server provided persistence across requests. While these are rarely*
- > *significant issues unless the client is a hand-held accessing via a*
- > *dial-up, they are still pure client problems.]*
- >
- > *BTW, if I had to deal with a server that was architected for stateless*
- > *data, I would be inclined to isolate that in the subsystem in the client*
- > *that talked to the server rather than modifying the way the problem was*
- > *solved to suite a stateless server. So that subsystem would understand*
- > *the data rules and provide caching for subsequent requests.*
- >

hmmm, not with this, my brain is wired the other way around, you need to explain in the other post an event driven world and map a simple

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

command driven model to an event driven one.

>>
>>
>>>
>>>>
>>>>>Essentially one has made every client responsible for also managing and
>>>>>optimizing DB access in addition to solving the customer's problem.
>>>>>That duplication fine if all the customer is doing is ad hoc reporting
>>>>>and data entry because it can be largely reused as in RAD IDEs. But
>>>>>when the applications are also solving significant problems, one has to
>>>>>touch /all/ those solutions when something changes in DB-land. That's a
>>>>>maintainability problem, which must be important because one is doing OO
>>>>>in the first place.
>>>>>
>>>>
>>>>
>>>>
>>>>>If you need to put stuff in the DB, I would put it in and keep that
>>>>>state in the DB server.
>>>>
>>>>
>>>>
>>>>>Then there will be a whole lot more physical I/O on the DB that makes
>>>>>the critical path performance bottleneck even worse.
>>>>
>>>>
>>>>>We talking past each other,
>>>>
>>>>>If i say keep the state on the client, you have a problem, if I say
>>>>>keep it on the server, you have a problem...I've missed something...or
>>>>>you have.
>>>>
>>>>>All I am saying is that there is no Free Lunch. Wherever one keeps the
>>>>>state there are potential problems. So there is no pervasive rule what
>>>>>state management one should employ; it is a fundamental architectural
>>>>>issue for Systems Engineering.

Then we agree = about that at least, though if there is no requirement from the clients perspective to persist data for any length of time past that of the direct interaction between client and server, I don't see the downside of it being on the client.

>>>>>
>>>>>>One can then allocate object state machines to event queue managers for
>>>>>>concurrency and put each event queue manager in its own thread. Simple
>>>>>>blocking constraints can even be supported via semaphores in the event
>>>>>>queue managers rather than invoking the <expensive> thread pausing
>>>>>>mechanisms.
>>>>>>
>>>>>>
>>>>>>
>>>>>>hmmm, can't quite see this.
>>>>>>

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

> >> Which part, blocking for data integrity or queue manager per thread for
> >> concurrency? (The answer might be lengthy to either, so I don't want to
> >> answer until I know which one.)
> >
> >
> > if "blocking for data integrity" means ensuring single threaded access
> > to internal state within an object i.e. lock(this) then you need not
> > explain...I understand this but dislike it as I see it as an
> > unnecessary overhead associated with usually unnecessary
> > multithreading and it's very easy to get deadlocks etc.
>
> No, I am talking about making sure concurrent threads don't step on one
> another. If a method in thread A is accessing a clump of attribute
> values one generally does not want thread B to be writing them
> concurrently. As a practical matter one can only deal with this at the
> level of method scope without turning one's mind to mush. One assumes
> all data accessed by a method needs to be consistent so one blocks that
> data from update during the method's execution.

i.e.

```
class CFoo
{
  void SetSomeState(int a)
  {
    lock(this)
    {
      x = a;
      y = x+1;
    }
  }
  void SetStateADifferentWay(int a)
  {
    lock(this)
    {
      y = a;
      x = y-1;
    }
  }
}
```

now I've read below, i.e. the same thing in a different manner.

>
> There are a couple of ways to implement blocking. One is a semaphore at
> the instance level that is checked by the event queue manager. If it is
> set the queue manager can't pop an event addressed to that object. So
> one sets the semaphore in the objects whose data will be accessed prior
> to consuming the event for the method in hand. [One can't consume the
> event for the method in hand unless all the semaphores are unset. So
> there is some Mickey Mousing to be done to deal with polling/setting

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

- > *multiple semaphores. This is the price one pays for much finer control*
- > *at the instance level.]*

Sounds dreadful – I'm not saying it's a bad idea, just not immediately attractive.

- >
- > *Another way is for the event queue managers to block one another. That*
- > *is, one pauses the queue manager controlling objects that may modify the*
- > *data from popping events. This is equivalent to pausing the entire*
- > *thread where the data may be updated. Typically this is done by the*
- > *queue managers sending each other events with a PAUSE/PAUSED protocol.*
- > *This is nice and simple, but in practice it tends to trash concurrency*
- > *unless there are a lot of threads (e.g., each object state machine has*
- > *its own event queue manager) because most methods access data attributes*
- > *and the entire thread is paused even if the next popped event goes to an*
- > *object whose methods don't update the relevant data attributes.*
- >
- > *Either way one needs a dependency map of what methods update data*
- > *attributes in other objects. Fortunately that is a static map that a*
- > *code generator can routinely generate. Though the infrastructure is a*
- > *bit tedious, it is very aspect-like and, consequently, cookbook. It*
- > *will generally be substantially more efficient than using the OS thread*
- > *pausing infrastructure because that has to deal with instruction-level*
- > *context switching.*
- >

Hmmm, maybe I'll just stick to a single thread.

- > >
- > > *I also implement a single threaded queue manager and push my*
- > > *environments multithreaded events into it.*
- > >
- > > *what I don't understand is your use of state machines.*
- >
- > *Each object's behavior is described with a state machine. These are*
- > *real states machines (as opposed of GoF State patterns) that have true*
- > *asynchronous events that are enqueued in an event queue manager. The*
- > *equivalent of a single threaded subsystem or application is having a*
- > *single event queue manger for all objects that does not pop an event*
- > *until the action for the current event completes. However, one can have*
- > *multiple event queue managers assigned to specific objects or groups of*
- > *objects. If one then assigns the event queue managers to different*
- > *threads, one can have concurrency as the event queue managers pop events*
- > *in parallel.*
- >

Hmmm, this is not OOP stuff is it.

- > >
- > >

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

> >>>
> >>>>[If blocking is necessary, it has to be managed somehow. Doing it by
> >>>>pausing state machines, though, is more aspect-like (i.e., the code
> >>>>looks the same everywhere) and is easier to manage because the scope is
> >>>>at the state action (method) level and one can count on the FSM rules to
> >>>>ensure that scope.]
> >>>>
> >>>>
> >>>>
> >>>>interesting but not with it.
> >>>
> >>>>Do you mean I didn't explain it well or that it is irrelevant to your
> >>>>applications?
> >
> >
> > I don't understand it (I do know what a FSM is but not it's
> > application in this context), I need a very simple example e.g.
> > something trivial like.
> >
> > class CFoo
> > {
> > private int x;
> > private int xPlusOne;
> >
> > void SetX(int y)
> > {
> > x = y;
> > xPlusOne = y+1;
> > assert(x + 1 == xPlusOne); // sometimes this fails.
> > }
> > }
> >
> > how do you prevent this using a FSM.
>
> You don't. This is a knowledge accessor. One only describes behavior
> responsibilities with state machines.
>
> More to the point, it is really an implementation issue that depends
> upon how 'int' is defined and what the value of 'y' is. That's an OOP
> tactical issue rather than a state machine flow of control or
> encapsulation issue.

Sorry I didn't make it clear, it fails in a multithreaded environment
because if two threads access the first thread gets to (say y = 1 for
this one)

```
x = y; i.e x = 1  
xPlusOne = y+1; i.e xPlusOne = 2
```

then the second one gets in with y=2
x = y i.e. x = 2

comp.object: Re: What doesn't lend itself to OO?

then the first thread goes
assert(x + 1 == xPlusOne); and BOOM

this is a trivial case, yet how is it handled in your world?
If not FSM then.....

>
> >
> >
> >>>> *We seem to have different views of COM. B-) To me the primary
> >>>>mechanism of COM composition is mixin-like inheritance. In one wants to
> >>>>add a streaming facility to the component in hand, it is inherited from
> >>>>Streamable (or whatever it is called -- it's been a long time since I
> >>>>was exposed to COM and I wasn't paying much attention even then).*
> >>>>
> >>>>
> >>>>*How would you do it in C++. If you would use MI in C++ then you can MI
> >>>>the interface and message forward (weak delegation), if you would do
> >>>>it another way, and I expect you would, the COM would allow you to do
> >>>>that.*
> >>>>
> >>>>*The way I see COM is that it is equivalent to C++ MI of implementations.
> >>>> I am not a fan of MI on cohesion grounds (Printable Sony Walkmans).
> >>>>Nor am I a fan of implementation inheritance; it seemed like a good idea
> >>>>at the time but opened a Pandora's Box of foot-shooting.*
> >>>>
> >>>>
> >>>>*But as I say it does not I believe encourage this any more that MI of
> >>>>completely abstract base classes.*
> >>>>
> >>>>*When I compose new widget X from widgets A and B, X gets the intrinsic
> >>>>behaviors of both A and B (i.e., X has no intrinsic behaviors prior to
> >>>>the composition). The mechanism of the composition is MI because my new
> >>>>widget is inheriting properties for both A and B, which are independent
> >>>>entities. It is implementation inheritance because X doesn't provide
> >>>>the default implementations; those are provided by A and B. The only
> >>>>way X gets a unique implementation is if I specifically overrides the
> >>>>implementation provided by A or B.*

Either I don't understand you or I don't understand COM or you don't understand COM.

in VB6 speak in C++ style psuedo OOP (i.e. COM) I'll drop explicit interface definitions....

```
class CWidgetA
{
    public void Widge() {...}
}
```

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

```
class CWidgetB
{
    public void Wodge() {...}
}

class CWidgetX
{
    private CWidgetA a = new CWidgetA(...);
    private CWidgetB b = new CWidgetB(...);

    public void WidgeWodge()
    {
        a.Widge();
        b.Wodge();
    }
}
```

There is NO MI, this code could be implemented in C++,C#,java,VB6 (COM).

I could do

```
class CWidgetA
{
    public void Widge() {...}
}

class CWidgetB
{
    public void Wodge() {...}
}

class CWidgetX : implement CWidgetA,CWidgetB
{
    private CWidgetA a = new CWidgetA(...);
    private CWidgetB b = new CWidgetB(...);

    public void Widge() {a.Widge()}
    public void Wodge() {b.Wodge()}
}
```

But again this could be implemented in C++, C#, java, VB6 (COM)

There is nothing more VB6 about this code that any of the others.

>
> >>>>>>> *In contrast, individual applications are abstracted to a very particular*
> >>>>>>> *problem in hand. So if one routinely applies the COM composition*
> >>>>>>> *paradigm one ends up bypassing OO maintainability in favor of the view*
> >>>>>>> *that it is easier to cobble together a new component for a specific*
> >>>>>>> *change than to worry about modifying the existing application to be in*

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

> >>>>>>synch with the problem space. The result is enormous code bloat.
> >>>>>>Worse, it leads to architectural drift in the applications away from the
> >>>>>>problem space leading to a maintainability nightmare when the
> >>>>>>application matures.
> >>>>>>
> >>>>>>
> >>>>>>
> >>>>>>Bad implementation does not make a bad tool. COM may encourage this
> >>>>>>because it encourages composition over inheritance, there is less
> >>>>>>coupling within an application and so it becomes much easier to coble
> >>>>>>a bodge together than have to reengineer the system due to a new
> >>>>>>requirement.
> >>>>>>
> >>>>>>And when one has collected together a flock of
> >>>>>>almost-but-not-quite-the-same widgets and the requirements change, how
> >>>>>>does one decide which widgets need to be modified or replaced? One can
> >>>>>>guess that only a specific context is affected and modify that. Then
> >>>>>>one must have a whole lot of faith in one's test suite.
> >>>>>>
> >>>>>>
> >>>>>>Again to "... collect together a flock of
> >>>>>>almost-but-not-quite-the-same widgets", would not be good practice, to
> >>>>>>have them available is nice and better than not having them available.
> >>>>>>
> >>>>>>You are losing me here. The only way to avoid that is by modifying the
> >>>>>>existing widget for the new requirement rather than composing a new,
> >>>>>>very similar COM widget. But then one is back doing what you argue COM
> >>>>>>avoids -- re-engineering the application. That's because the only way
> >>>>>>one widget can serve all contexts is by ensuring all contexts are
> >>>>>>consistent with the requirements change vis a vis side effects.
> >>>>>>
> >>>>>>
> >>>>>>
> >>>>>>We missed each other -- I thought you were talking about purchasing and
> >>>>>>using at the same time components from a 3rd prty that do almost the
> >>>>>>same thing.
> >>>>>>
> >>>>>>But I still don't agree, I see nothing in COM that makes the
> >>>>>>production of multiple very similar widgets any more likely than in
> >>>>>>C++.
> >>>>>>
> >>>>>>That's probably because COM was at least partially implemented with C++.
> >>>>>>B-) The issue is not about what one CAN do; it is about what one
> >>>>>>SHOULD do.

No it's not.

Your contention is that COM is bad because it encourages the MI of interfaces.

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

We both agree that MI is in general dangerous and should be the exception rather than the rule.

I have shown you that good practice is as easy in VB6 as in C#, java, C++ and in fact is easier because you cannot go even further and walk the tightrope of true implementation MI.

I can produce a bad implementation in VB6, but one can do so in C++,C#, smalltalk, java, C, assembler, Eiffel, CLOS.....anything you choose to give me, I can produce something like an AppleOrange.

- >
- > *My problem with COM is that it makes it easier to build a new widget*
- > *than diagnose and modify an existing widget.*

Why?

- > *That's fine in certain*
- > *contexts like providing very generic layer interface infrastructures --*
- > *in fact, creating new widgets on an ad hoc basis is exactly what one*
- > *wants to do. The problem is that it also allows one to do the same*
- > *thing in other contexts where it is not appropriate.*

how does it do this over and above any other mainstream OOP?

- >
- > *Consider an original application that satisfies the requirements with*
- > *widget X that is composed of {A, B, C}. Now the requirements change and*
- > *a new context is introduced where one needs basically the same X except*
- > *that one needs a slightly different behavior for C. In the COM paradigm*
- > *the natural thing to do is make a copy of C, modify it to C' and then*
- > *create a new X' that is composed of {A, B, C'} and use that X' in the*
- > *new context while leaving the old X and C versions for the old context.*

A ha, now we have the meat.

You can choose to do this if you so wish, but you don't have to, it provides a strict versioning policy where any change indicates a new component BUT it says nothing about how versioning should be carried out in practice. I *much* prefer there to be 2 logical widgets that are different than 1 logical one with two different implementations.

You are describing a problem that dogs all software and a solution that *may* be carried out in COM if you so choose.

- >
- > *That leads to bloat because of the parallel existence of {X and X'} and*
- > *{C and C'}. But in a large application contexts with volatile*
- > *requirements is also leads lots of maintenance problems in managing all*
- > *those not-quite-the-same versions as they proliferate. Worse, it leads*
- > *to architectural drift because the new widgets are created myopically;*

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

- > *they are focused in resolving specific contexts in the easiest fashion*
- > *possible.*

It's generally bad practice in any language using any infrastructure, I have never seen anyone even suggest doing this in the context of the production of a single version of a piece of software, only ever in the context of a new version that requires backwards compatibility where deployment of new clients is prohibitive in some manner, and only then as an option...

i.e.

- i) modify and publish new widget, leaving old one alone
- ii) reengineer
- iii) withdraw support of old interface.

If you choose i) that's up to you, but I don't see how COM 'encourages' this any more than C++ or all the others.

- >
- > *In contrast, in OO A, B, and C would be abstracted from problem space*
- > *entities individually and they would collaborate in a peer-to-peer*
- > *fashion. When requirements introduced a new context where C needed a*
- > *new behavior, that would be resolved within the existing context of C's*
- > *implementation and its collaborations. More important, it would be done*
- > *while ensuring a problem space mapping that reflected how the new*
- > *context was integrated in the problem space itself.*

And what prevents you doing this and then implementing in COM?

- >
- > *While that Big Picture view is probably not as easy to manipulate as the*
- > *myopic COM approach in many cases, it preserves the long-term stability*
- > *implicit in ensuring good problem space mapping.*

Again I think you confuse COM with some COM code that you have seen.

- >
- > >>>>*But the real issue here is architectural drift. That increases the*
- > >>>>*chances that a relatively minor change in the problem space will trigger*
- > >>>>*a massive refactoring of the application. That sort of refactoring is*
- > >>>>*independent of whether COM techniques were used; is a direct result*
- > >>>>*of the software structure not matching the problem space structure,*
- > >>>>*regardless of the construction techniques. My point is the the COM*
- > >>>>*composition paradigm strongly encourages architectural drift. IOW, it*
- > >>>>*may be somewhat more difficult now to implement an individual change*
- > >>>>*while preserving the problem space structure than doing so while*
- > >>>>*ignoring the problem space structure, but one will pay for it later.*
- > >>>>
- > >>>
- > >>>
- > >>>*Again, it encourages decoupled code, and this can be abused – but I*

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

> >>> *would hope that you were a champion of high cohesion and low coupling,*
> >>> *we can if you want return to the days of deep inheritance tree's where*
> >>> *every small shift in requirements triggers the avalanche – I prefer*
> >>> *that my architecture to be robust and flexible enough to absorb many*
> >>> *shifts before it collapses around me.*
> >>
> >> *Alas, I see COM objects developed for a specific application as the*
> >> *antithesis of high cohesion! I agree COM reduces inheritance breadth*
> >> *effectively to one level. But that first level is a killer because it*
> >> *is arbitrarily broad due the MI of composition. More important, that MI*
> >> *is completely arbitrary; there is no notion of the cohesion resulting*
> >> *from problem space abstraction around unique and well-defined entities.*
> >>
> >
> >
> > *I see objects developed for a specific application as the antithesis*
> > *of high cohesion, but I see no need to specifically insert the word*
> > *COM in this sentence.*
>
> *When one forms a COM component via composition one merges other COM*
> *entities with quite disparate properties. The new entity has both sets*
> *of properties. Since those properties are logically unrelated, that*
> *necessarily represents a lack of cohesion in the new entity. Moreover,*
> *since the merging is done for the benefit of the developer in dealing*
> *myopically with the context in hand, there is no attempt at maintaining*
> *the cohesion implicit in the problem space. (Perhaps more precisely,*
> *one mixes problem spaces in the same entity by doing things like making*
> *a Customer persistable.)*
>

Maybe I don't understand your notion of composition

> From <http://c2.com/cgi/wiki> this is composition

```
class Person {
public:
    void eat(Food bite) {
        itsMouth.eat(bite);
    }
private:
    Mouth itsMouth;
};
```

and this can be done in VB6, but I see no notion of mixing problem spaces?

If you don't like the VB6 standard libraries or any other implementation in COM then fine, that is a different issue.

COM is simply a specification of an infrastructure for interfaces...nothing else.....

Re: What doesn't lend itself to OO?

comp.object: Re: What doesn't lend itself to OO?

There are implementations of COM authoring tools like VB6 which you may not like, but I see nothing in your argument that says the ability to declare interfaces and implement them in classes, or the implementation of that in VB6 is bad.