Re: What doesn't lend itself to OO?

Source: http://coding.derkeiler.com/Archive/General/comp.object/2004–08/1026.html

---

From: H. S. Lahman (*h.lahman_at_verizon.net*)
Date: 08/07/04

```
Date: Fri, 06 Aug 2004 23:39:43 GMT
```

Responding to Nicholls...

Something went awry in Netscape and I lost my original response, so this
is a second try...

>>>>>// 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 indoctrunated),
> depending on the configuration data associated with CGMTClockService,
> effictively 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.

I don't see those responsibilities in a single object. For example, talking to a network is complex with its own paradigms (e.g., proxies). Ideally one wants to isolate that stuff in a layer or subsystem. More important, one wants to reuse it for all distributed messages employing a given protocol. That is, one does not want to have to duplicate the implementation details in other objects, like CGMTWeatherService. In addition, one wants to be able to swap network environments without touching anything in the problem solution.

What COM is doing is providing that entire layer and hiding it from you. That's convenient because you don't have to write it even once. The downside, though, is that COM implements it one way with one view of how it is used. IOW, its interface is designed around the COM designers' view of how the network will be used. One way that is manifested is in constructing CGMTClockService via composition of the network interface. That technique is very convenient from COM's viewpoint but it isn't very OO because it trashes encapsulation by introducing external issues into the application's implementation objects.

In particular, the application shouldn't need to know that the service is distributed at all. The application's messages should be exactly the same where the service is on another platform or linked into the same process. So there should be no need for a factory in the application that needs to know about whether a proxy will be used. But in the COM world there must be because that is a local optimization that COM can't anticipate for you. So the application must do it and that means the internal application implementation needs to know too much about the outside world.

What happens when one decides to change the network protocol? There is no way to do that without also touching the object with clock semantics (CGMTClockService) in the application itself. In addition, one may have to touch other problem solution objects (e.g., factories) that talk to it. Regardless of how elegant the composition techniques are, that still means there is risk of breaking something in the application solution. Therefore one must functionally test the problem solution itself completely even though theoretically it has not been touched.

OTOH, OO encapsulation doesn't have a problem here. The same object will not encapsulate both clock semantics and network semantics. More important, that encapsulation applies at the subsystem or layer as well so that no object within the application solution will even know that the network exists. So no object within that interface will be touched and one can be completely confident that the solution behaves exactly the same way.

This is all pretty obvious in a case like a distributed boundary. However, the same thing applies locally when one uses composition to construct objects that have disparate responsibilities using composition. One can't modify one aspect of the composition without risking breaking the other aspects or modifying how other objects

interact with the object. OO encapsulation is about avoiding that risk
and composition increases it.

Note that it is good OO practice to keep subclassing trees simple. One
reason for that is cohesion. All instances of the tree are members of
the root superclass. If there are lots of complicated specializations,
/all/ of the those properties are represented in the root superclass.
Complex subclassing trees almost guarantee a lack of cohesion in the
root superclass' properties. And that is with specialization rooted in
a single conceptual entity. How much easier is it to lose cohesion when
one is composing from multiple, disparate conceptual entities?

\>
\>
\>>>
\>>>>*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.*

OK, we have a small disconnect here. When I think of 'proxy' I think of
it in the more restricted sense of networking. IOW, a network proxy
would be external to the application process, not in it. You seem to be
using it here in the more general sense of a surrogate (i.e., it is the
application's view of the service). If that is the case, I would see it
simply as a Facade that relays the message to a networking subsystem
interface that, in turn, relays it <eventually> to the service's
interface that, finally, dispatches it to a CGMTClockInstance in the server.

A bigger disconnect seems to about the clock service example itself. I
may be taking it too literally and focusing too much on how it would be

implemented. For example, I see no reason for the service to care
whether the client is there. The clock is part of its intrinsic service
and would always be implemented so long as the service was up. It would
offer the GetTime(zone) service to all clients whenever it was available.

[This probably also reflects on the statelessness issue. I would expect
the service to have a single instance of Clock to service all clients so
there would be no need to optimize stateless objects.]

>
> *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.*

Alas, .NET was still mostly vaporware when I retired and I know zilch
about it. [And as a translationist, I didn't care anyway. B−)]
However, this sounds like .NET provides more substantial surrogates that
–– via composition –– hide the networking details to a considerable
extent. For example, how one instantiates CGMTClockService on the
client side will determine whether a real network proxy will be employed(?)

>
>
>>*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.*

Let's not get into type systems. That's a 3GL issue and my concerns
here are at a more fundamental OOA/D level.

>
> *hmmmmmm, I'll ramble a bit....*
>
> *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.*

Right. This is entity identity. But, unlike Data Modeling, it does
have to be explicit. For example, identity can be maintained through

relative position in an ordered set or through relationships.

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

I see this as type identity rather than entity identity. It only needs
to be explicitly defined at the 3GL level and then only for error
checking. The type identity, like entity identity, is handled at the
OOA/D level through relationships. If one needs a specialization, the
relationship is instantiated only to subclass instances. If the
relationship navigated is instantiated to the superclass, one is
indifferent to specializations and one accesses the general properties
polymorphically.

>
> *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*

As I indicated later in the message, I see stateless objects as an
implementation strategy to avoid the first situation _when it logically
exists_.

>>>*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.*
>>
>>*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 simple 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.*

I would argue that I am describing an OOA model, not an implementation. Back to my point above, I see having a single clock for all clients as a fundamental problem space solution to the clock service requirements. That is, I see the clock example as a poor example of statelessness issues because there would be no requirement that would need a clock per client.

I contrast that with something like a POS ShoppingCart. Now there is clearly a requirement that one be able to associate particular state variable values with particular <ephemeral> customers. That's a resource issue on the server if the state is maintained there. But at the OOA level one logically has a shopping cart per client and each shopping cart has unique state data.

One can get around that problem _in the implementation_ by introducing a stateless shopping cart in the server that has only behavior and the relevant state variable data is passed with each request as part of the message data packet (or is stored in the DB).

>
> *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.*

I see this as a special problem because of the nature of network communications. Somebody has to keep track of connections, ports, addresses, etc. and that does imply some sort of identity being associated with clients. However, I also see that as a separate design problem from the semantics of DB communications between client and DB. IOW, that software solves a different problem than the one that is solved when processing individual queries.

I would argue that one big reason for isolating it is that statefulness is necessary because one must <somehow> track entity identity from messages to connections. If one isolates that statefulness from the routine query processing, one can optimize it to minimize resource usage –– very likely not in a very OO way (e.g., an array of C++ structs rather than individual objects) simply because of the very low level optimization issues.

OTOH, the semantics of query processing is quite a different thing. If
the middleware needs to process the data on the server between client
and DB, then one is potentially back to the POS shopping cart issue
above and one faces an implementation optimization decision about
whether to employ stateless objects for the server implementation.
Logically, though, an OOA of the server middleware processing would
still have stateful objects; the conversion to stateless objects would
be a mechanical conversion at the low–level OOD or OOP level. (Alas, as
we've already discussed, that will affect the interface if it isn't
stored in the DB but it can be encapsulated on the client side.)

\>
\> *You may well be decribing how you work, but you may be more*
\> *enlightened that 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.*

Only if all the server is doing is reading/writing the DB via SQL
queries. SQL itself is essentially providing the decoupling. However,
if the server is "fat" and does something with the data before it is
written (e.g., checks stock, computes discounts, etc.), it will likely
need more semantic knowledge of the context.

[FWIW, I think this is analogous to using COM objects to provide glue
for distributed processing. For any serious problem solving one does
not want SQL statements littering the code. So one isolates the SQL to
a subsystem that exists solely to communicate with the DB. That
subsystem decodes/encodes the SQL datasets from/to the formats that are
best suited to solving the problem. Similarly, one does not want COM
objects composed with distributed support in the application solution;
one wants simpler objects tailored to the problem in hand.]

\>
\>
\>>*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 it's 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
> RDB. How do you manage the lifetime of this session? does it last
> foreverm, 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.

The normal solution is the RDB equivalent of garbage collection. A
separate process goes through those session tables and deletes entries
whose time stamp is out of date. [When a session completes with the
client issuing the equivalent of a commit request (e.g., go to
checkout), the temporary session data (e.g., shopping cart) is read,
written to a permanent location (e.g., an Order), and deleted from the
session tables. If the client was on vacation and issues another
request after the garbage collection the server read fails and a time
out message is returned to the client.]

The client side equivalent for something as trivial as a POS order entry
would be to keep a cookie (shopping cart) on the browser that is
included with each request. Simpler client implementation and no server
state at the cost of more bandwidth.

>>>>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

Actually, I didn't know there was a CET. I've led a sheltered life on
my side of the Pond.

>
> 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.

I am curious about what the difference in behavior is. (I would expect
just an offset in the time –– like GMT zone X = CET zone Y + 4.)

>>>>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
>>>???
>>
>>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,*

Only if one defines 'mainstream OO' as what procedural converts to OOP
development do when they never bothered to learn what OOA/D is about.
B−)) That is basic OOA/D and it is the root reason that message is
separated from method. (In UML one can specify the object interface
separately from the behavior methods so it is possible to name the
object interface after the condition change and <internally> map that to
a method that describes what the object will actually do.)

Note that events are /always/ announcements; the R−T/E people figured
out I'm Done was the better way to go than Do This long before OO was
developed. The Do This paradigm led to hierarchical implementation
dependencies that became the root cause of the legendary Spaghetti Code.
One could argue that the primary OO advantage is eliminating
hierarchical implementation dependencies through encapsulation,
implementation hiding, cohesion, peer−to−peer collaboration, and −− most
important of all −− conceptual separation of message and method.

The fact that OOPL compromises with the Turing/von Neumann computational
models have removed that separation for invoking methods does not change
the fact that they are conceptually separated _when designing objects
and collaborations_. [OOPL designers will be quick to argue that
message and method are still separated. The message is the method
/signature/, which is a different critter than the method
implementation. I find that a bit pedantic.]

The use of procedural message passing is benign so long as the caller
was written without any expectation about what the receiver will do.
That comes for free if the call is conceptually regarded as an
announcement. That is, the sender has done something the receiver needs
to know about and the receiver's method call signature is the message
the receiver wants used when that happens.

When using procedural message passing the message identifier must be
named for either the change in sender state OR what the sender does, not
both. It is merely a 3GL naming convention to do the latter and that
convention should have nothing to do with the way one constructs the
software.

One only gets into trouble if one maps a procedural approach onto the
OOPL message passing and treats method invocations as imperatives. That
becomes instantly apparent when an OOPL program has behavior methods
that return a value. That creates an implementation dependency in the
caller on what the responder does and it will get a good OO reviewer to
bring out the crucifixes and garlic cloves immediately.

[Since OOPLs employ procedural message passing for both behavior and
knowledge access and getters/setters must return values, there is a
strong temptation for naive converts to OO development to continue

employing the procedural tradition of having the responder do something the sender needs done before it can continue. But in OOA/D knowledge and behavior are quite different things and one designs around them quite differently (e.g., knowledge access is always treated as synchronous while behavior access is always treated as asynchronous). In effect, they are apples & oranges and one needs to understand that when using the OOPLs where the distinction does not exist.]

Note that one common test of proper encapsulation and context independence is to reorder the calls to other objects' behavior methods. If the behavior of the method in hand remains the same under unit test, there is no hierarchical dependence on what the called behaviors do and the calls are announcement messages. (In practice this is almost never done literally but it is a quite common mental test that reviewers apply.)

Though rarely done in practice, it is quite possible to write all the code for all object behavior methods without including any calls in them except knowledge getters/setters. One can even unit test them at that point, though there are practical reasons for knowing that the method issued the correct messages. One can then make a second pass where one adds behavior method calls to the end to those methods. For example, one might do a UML Interaction Diagram to define collaborations and then back−track to the methods to add the calls from the Interaction Diagram. The point here is is that OO behavior collaborations are defined at a different level of abstraction than individual object methods.

In addition, in a well−formed OO application one can always fully unit test an object without implementing any other object behaviors and without employing dynamic stubs in the test harness. [Dynamic stubs are unnecessary because OO behaviors don't return values for the caller to use so all one needs to know is that the method was called with the right arguments (i.e., the message was sent).] More important, because dynamic stubs are unnecessary, one has complete confidence that the object will behave in situ exactly as it did in unit test.

Finally, one can apply exactly the same sort of DbC precondition to postcondition mapping that R−T/E people use for generating events for interacting state machines even when one is not using state machines. The precondition of executing a method is still the postcondition of some other object method's execution. Therefore one invokes the method in hand in the method where that postcondition prevails. Which segues to...

>
>
>>*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.

From the above, my assertion is that the 'command perspective' is
merely an accident of 3GL naming conventions. From an OOA/D viewpoint
there is no difference between generating an event and invoking a
behavior method –– if the objects and collaborations were designed in an
OO fashion. Alas, if there is a distinction it exists because one is
mapping a procedural construction paradigm (e.g., SA/SD) onto OOP.

>>>>>>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.
>>>>
>>>>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;
> }
> }
> }
```

Close, but not quite what I had in mind:

```
Class CFoo
{
private:
    int x;
    int y;
    bool isLocked;
    ...
public:
    setX (int v) {x = v;}
    setY (int v) {y = v;}
    int getX () {return x;}
    int getY () {return y;}
    bool getIsLocked () {return isLocked;}
    lock () {isLocked = TRUE;]
    unlock () {isLocked = FALSE;}
    setSomeState (int a) {yours w/o lock()}
    setStateADifferentWay (int a) {yours w/o lock()}
    ...
}

CBar1::doIt ()
{
    ....
    if (myFoo–>getX () > myFoo–>getY ())
      ...
}

CBar2::doIt ()
{
    ...
    myFoo–>setX (42);
    ...
```

```
    myFoo−>setY (41);
    ...
}
```

One can't have CBar2::doIt changing the values as CBar1::doIt is executing. However, it is insufficient to have CBar2::doIt invoke myFoo−>lock when it starts executing and invoking myFoo−>unlock when it completes. Nor can CBar1::doIt call myFoo−>getIsLocked around its call to the getters.

There are several reasons: code clutter, a polling loop if Foo is locked when CBar1::doIt is called, and general race condition safety. So one invariably builds everything into the event queue manager code. The queue manager does the checking and locking of Foo before popping the call to the method that will actually do the read (CBar1::doIt). The queue manager also takes care of locking Foo when CBar2::doIt is about to be popped.

One doesn't need to do Foo::lock in setSomeState or setStateADifferentWay as you did. They, by good OOA/D practice, are invoked synchronously from some other object's behavior method because they are simply knowledge accessors. (If they aren't, then they would be state machine actions themselves.) It is that method (the analogue of CBar2::doIt) that will trigger locking of Foo. That would also apply to any behavior method of Foo that modified its state data. IOW, everybody is processed exactly the same and the locking code is isolated to the event queue manager.

>>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
>>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.*

As I indicate above, it is not that bad because everything can be isolated in the event queue manager. My example above oversimplifies a bit. Most likely there would be no getIsLocked because that would require some sort of two−phase commit to avoid race condition problems.
 Instead the queue manager would invoke Foo::lock directly and it would return success (locked) or failure (already locked). But it is still pretty simple code.

[Before you jump all over me for a behavior method returning a value,
Foo::lock isn't a behavior responsibility; it is a knowledge (isLocked)
accessor. B−)]

The only prolific complication is adding the lock/unlock methods, but
that is a pretty simple aspect−like support of concurrency and the code
will be exactly the same for everybody (one could inherit is from some
MotherOfObjects). The real problem is the mapping in the event queue
manager of events to the data that they access in order to know who to
lock. As I indicated in the previous message, that can be reduced to
effectively a table lookup; the tricky part lies in initializing the
table. That's easy and reliable for an automatic code generator to do
but it will be nasty to get right for manual development. OTOH, those
kinds of concurrency issues have to be addressed somewhere, which is why
R−T/E people tend to limp a lot.

>>*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.*

Actually, I would argue it is quintessential OOP. Interacting object
state machines /enforce/ a lot of good OO practices because that is in
the nature of the rules for FSMs. In particular:

−− message (event) and method (state action) are clearly separated.

−− behavior collaboration is asynchronous, as it is in the OOA/D.
(Synchronous collaboration is just a special case of asynchronous where
the order of external events is predefined.)

−− methods are context−independent because a state cannot know what the
next or last states are so there can be no dependence on external
sequencing of operations in the action implementation.

−− object state machines provide excellent encapsulation because events
decouple their implementation from the outside world and because an FSM
is, by definition, self−contained.

−− States and transitions map very nicely into the notion of a problem
space entity having a life cycle.

-- Transitions are a very efficient means for enforcing constraints on
sequences of activities. (Efficient because no dynamic code is
executed; the enforcement exists in the static STT lookup.) More
important, they are very effective in doing so. Getting interacting
state machines to work (i.e., no "can't happen" events are consumed) is
nontrivial but once they do work they tend to be quite stable and reliable.

-- States represent conditions where particular rules and policies
apply. That makes it very easy to apply DbC in the form of precondition
vs. postcondition mapping, as the R-T/E crowd did long before OO.

-- Corollary: events represent pure peer-to-peer collaboration because
they are generated exactly where the condition prevails and they are
consumed exactly where that condition matters.

-- Since state machine "state" is quite different than state variable
"state", one is forced to think of them separately -- which is crucial
to managing persisted state in an OO fashion. (The alternative is
functional programming where one eliminates persisted state entirely.)

```
>
>
>>>
>>>>>>[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
>
> 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.....

OK. The answer depends upon whether SetX is is a knowledge accessor or
a behavior responsibility. Let's first assume it is a behavior so it is
in an action of an object state machine. In that case there is no
problem because one of the rules of FSMs is that only one event can be
consumed at a time. Therefore even if there are several (reflexive)
events on the queue, they will only be popped one at a time. So this
would only be a problem if one did NOT employ an event queue manager.

The second situation is that SetX is not in a state machine at all.
Instead, it is a knowledge setter. In that case, it will have to be
invoked from two actions in different state machines and my discussion
of blocking above applies. That is, one will have to ensure that the
second action's event is not consumed until the first action has completed.

If all behavior is in state machines AND one is using event queues, then
the blocking can at least be isolated to the event queue manager rather
than being incorporated into the individual objects. One also gets to
convert a dynamic problem (the rules and policies of who needs to be
blocked in terms of encoded IF conditions) to a static one by providing
the lookup table for what needs to be blocked.

>
>
>>>
>>>>>>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.

I thought I mentioned I have very little experience with COM and most of
my knowledge comes from the Executive Summary level (plus second hand
from a different group in our shop that was heavily into it), so the
last is a good possibility.

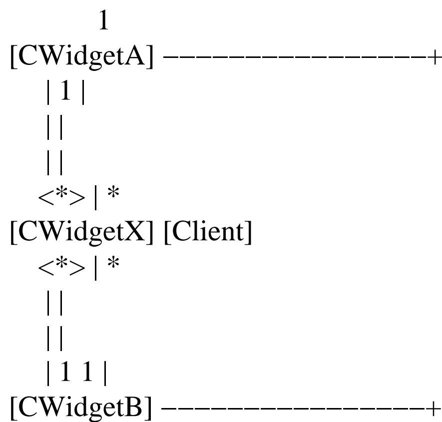
>
> in VB6 speak in C++ style psuedo OOP (i.e. COM) I'll drop explicit
> interface definitions....
>
> class CWidgetA
> {
> public void Widge() {...}
> }
>

> *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).*

True. But it is not very good OO code. There are two problems. First, CWidgetX is acting as middleman between the Client and CWidgetA and CWidgetB. That is, it is CWidgetX that decides how they should collaborate. This will be especially clear if there is a dependency between a.Widge and b.Wodge (i.e., a.Widge updates a state variable that b.Wodge needs to access).

In that case CWidgetX is defining the flow for control dependency (i.e., CWidgetA.Widge must be executed before CWidgetB.Wodge) _in its implementation_. So if the rules of collaboration between Client and CWidgetA and CWidgetB change, then one must modify CWidgetX's implementation. That is a major league OO no–no.

The second problem is related to the first: if Client needs to collaborate with a CWidgetA or CWidgetB instance it should do so in a peer–to–peer manner. Make no mistake, what you have here is:



The CWidgetX::WidgeWodge is just a euphemism for the combined calls to CWidgetA.Widge and CWidgetB.Wodge. More important, Client needs the specific specializations that CWidgetA and CWidgetB provide. Therefore, it needs to have direct relationships to them and Client should address messages directly to them via navigation of those relationships.

[Note that embedding objects is only one way for the Whole/Part disposition constraints implied by a composition association to be satisfied. (In UML a composition association is about Whole/Part, not the notion of composition we are debating.) It is very convenient if no other objects than the Whole need access to the Parts. However, if an external object does need access to the Part, then embedding in the Whole is not a valid implementation because embedding makes 'a' and 'b' part of the CWidgetX implementation, which should be completely hidden from other objects.]

Caveat: I am talking about problem space classes here, not the nuts & bolts computing space data holders like String, Stack, Array, etc. Or even classes like Matrix that are not specific to the problem space of the problem in hand.

```
>
> 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)
```

Same arguments apply. If the Client needs to talk to a CWidgetA, then let the Client do that directly. But there is potentially a third problem here: cohesion. In the first case the Whole/Part semantics could be argued as providing at least some logical cohesion. However, here the semantics of Widge and Wodge can be totally unrelated in the problem space and the composition can be solely for the convenience of

the developer.

However, I don't think these examples directly relate to the COM issues.
There one composes using an hidden implementation that is somewhat AOP–like. That implementation is not really a separate class; it is thoroughly integrated with the object in hand to provide an –able capability (e.g., persistable or printable). While that sort of composition is handy for hiding "canned" infrastructures, it isn't a good way to abstract most customer problem spaces.

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

While I think VB encourages a lot of very bad OO habits, I do