



search

email

50967 members! [Sign up](#) to stay informed.



Sponsored Links

Java and .NET
class-level
interoperability
Free eval of
JNBridgePro:
JNBridge.com

[Free VS Add-In](#)
for managing
projects, bugs,
features and
tasks. Download
a free copy now

[Free Trial!](#)
Map, generate,
and maintain
50% of your
.NET app code
Download now

Get the latest
licensing and
technical training for
.NET in the [Microsoft
US Partner Readiness
Center](#)

White Papers

[.NET Research Library](#)
Get .NET related
white papers, case
studies and webcasts

A SOA Versioning Covenant

[Discuss](#) [Printer friendly](#)



by Rocky Lhotka

A SOA Versioning Covenant

April 20, 2005

When it comes to distributed systems, versioning is always the Achilles Heel... One goal of service-oriented (SO) design is to help address the versioning issue by ensuring that every service and client of a service is an autonomous, independent entity. However, that's only part of the story.

What makes this even more interesting is that in an SO world versioning is merely a subset of a bigger issue. That issue is the fluid nature of the agreements (contracts) established between software entities. I may create a PostOrder service that accepts various *different* types of request, but always performs the same action. Then, I might version some of those request types. But versioning merely means that I have increased the number of overall request types – there's really no difference between a new version of a request and a brand new request.

Before I go any further, a quick disclaimer. In this entry I am talking specifically about SO design. I am *not* talking about n-tier or client/server design. They are totally different from SO. In an n-tier model, we're talking about an application that is designed as a set of interrelated logical layers. Those layers are then potentially deployed on different tiers (different processes or different machines). The fact that some layers of an application are physically separated from each other doesn't change the reality that these are just logical layers of a single autonomous entity.

In this article I am discussing solutions to some serious issues in the SO world. In most cases these solutions do not apply to the n-tier world, where good solutions already exist.

The Ups and Downs of Contracts

Modern thought around procedural, object-oriented and component-based designs tend to rely on strong typing. Strong typing is just another way of saying "contract". The whole point is that the caller knows all about the callee, and thus can optimize the call based on that knowledge.

Collectively we *love* contracts. They give us better performance, better debugging, better development experiences and more. Things like Intellisense and compile-time validation exist largely due to contracts. It would be easy to think that contracts and strong typing are nothing but good.

But there is a dark side. A formal contract comes with serious limitations. It can be defined as "a contract made binding by the observance of required formalities regardless of the giving of consideration" (dictionary.com). Consider what this says. It says that there is no flexibility, that the formalities must be observed no matter what.

Unfortunately this doesn't reflect reality. In the real world business or environmental needs cause software changes on a regular basis. Software changes imply changes to the provider, and that inevitably means changes to its "contract". A change to the contract makes it invalid, causing "renegotiation" with the consumer.

In plain language this means that contract-based programming is fragile and *will* break in the real world as software changes occur.

COM and CORBA were designed following this top-down contract-based worldview. The vision was

that we'd divine all the requirements a component would ever have, solidify those requirements into a formal contract and then program clients against the contract. A really cool idea in a static world, but pretty impractical in the wild.

What happened in the 1990's is that COM was "bastardized". It was altered to enable extensions to the contract without "breaking" the contract. And if the contract actually had to be broken we'd create a new contract (DoWork() and DoWorkEx() for instance). Eventually there were more contracts floating around than we knew how to handle. But even adding new parameters to an existing method actually broke the contract, since it required pre-existing clients to rebind to the new contract.

Kind of like when your credit card company changes the rules to raise your fees. You can opt out – but only by cutting up the card and never using it again. If you want to use the card, you must rebind to the new contract.

The Semantic Contract

When we talk about contracts most people think of the syntactic contract. This is the definition of an API or a message schema in terms of data types and method signatures.

However, there's also a *semantic* contract which is often overlooked. The semantic contract says that when I call DoWork() a certain set of things happen. This set of things includes the work being done, but also includes observable side-effects of the work and things like failure conditions. Though not documented or formalized in any way, the semantic contract is every bit as important as the formal contract expressed in IDL (or now in WSDL).

The semantic contract was a core contributor to DLL hell in COM and may very well cause the same issues in any SO application. The challenge is that a client *will* be programmed to work with (or around) the semantic contract. Often this is done through trial and error on the part of the client developer. Remember that there isn't really a "contract", so the only way to determine what actually happens when DoWork() is called is to do extensive testing and to program around the edge cases.

When DoWork() is altered later (and it will be) the contract is changed. Maybe the formal contract will change, triggering the client to rebind (and presumably retest, etc.). More insidious is the case where the formal contract *doesn't* change, because the client developer won't know to retest. Yet the semantic contract has changed out from under the client.

Note that I state that as a fact, not a possibility. It is simple fact that if DoWork() is altered in any way then its semantic contract will have changed. If DoWork() has *exactly* the same behaviors it did before the change, then it obviously wasn't changed now was it? If it was changed, then it has different behaviors in some way, and thus has a different semantic contract.

I confess, I'm being a bit trite here. It may be possible to alter DoWork() in ways that have absolutely, positively no externally visible differences. To preserve any side-effects, to throw the same exceptions, to include the same quirky bugs.

But that's rarely reality. Typically we're altering DoWork() to *fix* bugs, *eliminate* exceptions or *extend* side-effects. And in so doing we change the semantic contract.

In the mainstream use of DoWork() things will likely continue to function normally for our oblivious clients. But around the edge cases, where we fixed that bug, or eliminated that pesky exception things will break. Clients will fail and wonder why.

I say this with certainty, because this is *exactly* what happened over the past decade with COM and Win32. Entire companies to this day refuse to upgrade IE or install .NET because such installs might change MDAC. And changing MDAC might replace one of the core MFC DLLs, and that might stop various applications from running.

Did the formal contract of the MFC DLL change? Not at all – Microsoft preserved backward compatibility at the formal contract level very nicely. But the *semantic* contract is quite different and things blow up all over the place.

The big worry is that this will carry forward into the Web service, Indigo and broader SOA worlds. And it might. The same exact issues still exist.

Contracts in the SO World

A service (web or otherwise) has both formal (syntactic) and semantic contracts. Clients that use that service will be bound to a greater or lesser degree to those contracts.

In the service world there are two parts to the formal contract. There's the API and the message schema.

(Note that I'm talking about services as being messaging endpoints, not components-on-the-web. See

Related Links

Ads by Google

[Free SOA Software](#)

Award-Winning Web Service Broker Monitor & Manage Web Service Apps
www.actional.com

[Svc Oriented Architecture](#)

Build mission critical SOA. Download White Paper
www.iona.com/architectix

[Mainframe SOA](#)

Software to implement a Mainframe Service Oriented Architecture
www.neonsys.com

[SOA In-Depth Research](#)

Download Burton Groups in-depth technical SOA roadmap report
www.burtongroup.com

[Principles of SOA Design](#)

Download SOA Whitepaper Service Enable Your Organization
www.capeclear.com

my [TSS.net article](#) for more details.)

The API part of the contract is the method signature, like

Boolean = $f(\text{Integer}, \text{Integer}, \text{String})$

The service exposes one or more of these method signatures through its WSDL. The client binds to that as a formal contract. This is exactly like COM. Once you've got clients bound to your API you can never change it. That's why an API like the one above is bad. Need a fourth parameter? Sorry, can't have it.

Talking to the creators of technologies like Web services, WSE or Indigo you might hear that this is OK, because "behind the scenes" they build a message out of your parameters, so everything is still message-based. Which is true, but is rather meaningless. As long as our client code binds to this API contract we're stuck in versioning hell, regardless of whether the data is passed as a message by the infrastructure.

This isn't just a "services" issue. DCOM, RMI, IIOP and Remoting all take your parameter values and serialize them into a single "message" that is transferred across the network. This isn't anything novel or new, nor does it help address the drawback of being coupled to a formal contract at the API level.

Instead, you are better off sticking with more generic signatures, like

$f(\text{requestMessage})$

or

$\text{responseMessage} = f(\text{requestMessage})$

With this model the schema part of the contract becomes very important. The schema is the definition of the requestMessage and responseMessage portions of the above API.

Rather than restricting our API to only handle simple data types like Integer or Boolean, we work with flexible messages. The problem with using simple data types is that they aren't flexible. If you accept an Integer to start with, what happens if you later need to accept a DateTime?

But if you accept a requestMessage that includes an Integer, you can later opt to also accept a schema that includes a String. For instance, we might start with a requestMessage like this

RequestMessage -> Integer

Later we might extend the schema to accept both an Integer and String

RequestMessage -> Integer, String

Or we might opt to allow our method to accept two different schemas

RequestMessage1 -> Integer

RequestMessage2 -> String

Note what has happened here. We still have the pesky versioning problem, it has just been pushed down from the API into the schema.

Does this mean that the versioning problem is inescapable? Yes. Does it mean it will always be as bad as COM? No.

We have a much greater ability to deal with the issue at the schema level than we ever did at the API level. There are various design patterns that directly address issues around message versioning, and we can apply them in these cases to dramatically simplify the issue. With luck, our vendors will incorporate some of these patterns into their products so we don't need to worry about the issues directly.

Contracts or Covenants?

I'll get to some specific answers later in the article. First though, I want to propose something that is sure to be controversial. Specifically I want to suggest that the "contract" metaphor is wrong, and is leading us down a dangerous path.

Instead I suggest that "covenant" is a better metaphor. A covenant is an "if then else" agreement. If you do *this* then I will do *that*. It is a much more realistic expression of how software actually works.

The service is saying that "if you send me X, I will do Y". Implicit here is the ability for the service to

also say that “if you send me A, I will do B”, or even “if you send me Z, I will do Y”. Note that this directly allows us to have multiple covenants that result in the same action if desired.

This flexibility isn’t conveyed through the word “contract”, which is why covenant is a better term.

Services should be governed by a covenant. The service should agree that when we send it a message with a specific schema that the service will perform a certain action. Send it a different schema and a different action will occur.

This not only provides a better metaphor than the formal contract, but it also provides a mental framework for dealing with the semantic contract. And that is important. Nothing in the formal contract world even tries to address the semantic contract issue, and yet it is at the core of the versioning issue.

Note that I’m not suggesting that the formal contract on the API be changed. We need to retain that level of formality, because it provides the standard protocol by which we communicate with services. The contract remains

```
f(requestMessage)
```

or

```
responseMessage = f(requestMessage)
```

What I am suggesting is that requestMessage and responseMessage be governed by covenants rather than contracts.

Employing Covenants

Specifically I’m suggesting that any $f()$ should accept any message. It should then examine the message’s schema to determine its course of action. This is all governed by the covenants into which $f()$ has entered.

I’ll also suggest that there is one covenant to which all methods must adhere, the Exclusive Covenant. If $f()$ receives a schema for which it has no covenant, it will reject the message and take no other action. Methods are exclusive and only handle messages with acceptable schemas.

(At this point OO purists may see where I’m going. I’m suggesting a very traditional message-based world view along the lines of original OO thinking. A model where a message can be sent to any object at any time, and the object simply rejects those messages it doesn’t choose to handle.)

Let’s take a concrete example. Suppose I have a service that deals with customers, and it has an AddCustomer() method

```
responseMessage = AddCustomer(requestMessage)
```

We might define a covenant that says “if you give me a CustomerAdd message, I’ll add a customer and return a result indicating success or failure”. This is the *explicit* covenant. There’s also an *implicit* covenant that arises out of the very implementation of AddCustomer(). In its implementation it will have quirks, idiosyncrasies and edge cases that aren’t called out explicitly but which are very real.

CustomerAdd might look like this

```
<Customer id="" name="" phone="" />
```

and responseMessage might look like this

```
<Response success="" />
```

Clients can start using AddCustomer() based on this covenant. They might find quirks and edge cases implicit in its implementation and they will work around them. In short, they’ll become bound to both our explicit and implicit covenant.

Then we discover that some customers have two phone numbers. Now what? Now we establish a new covenant based on a new schema. “If you send me a Customer2Phones message, I’ll add a customer and return a result indicating success or failure”. Again we’ve created an explicit covenant. Note that the action we’re promising to take is the same. All we’re doing is extending our capabilities to accept different requests.

When we go to implement this new covenant, we must do so in a way that preserves the CustomerAdd Covenant as well. Adding a new covenant doesn’t mean that previous covenants can be discarded! In fact, what AddCustomer (or any service method) needs to be is a [Message Router](#). Behind the router are physical implementations of each covenant.

So planning ahead, when we implemented AddCustomer, it would have been built as a router that rejected all but the CustomerAdd schema. Messages of that schema were routed to a CustomerAddImplementation.

Now that we are honoring a second schema, the AddCustomer router will route CustomerAdd messages as before, and will also route Customer2Phones messages to an implementation of our new covenant: Customer2PhonesImplementation.

This design helps protect not only our explicit covenant, but the implicit covenant as well. Adding Customer2Phones may have absolutely no impact on the CustomerAdd behavior, and this design helps make that possible.

Of course in a real application we may do deeper changes, such as changing the underlying database schema or other technologies. This may impact existing implicit covenants in ways that are hard to predict. Unfortunately this is simply a basic fact that we must deal with over time. It is not possible to escape the fact that semantic behaviors change over time. However, this fact shouldn't stop us from trying to minimize the potential semantic changes that we *can* control.

If you stop and think about what this design does to the contract model, you'll realize that it shifted the contract to the implementation level.

```
Client->AddCustomer- /->CustomerAddImplementation
                    \->Customer2PhonesImplementation
```

Any given client now has a covenant, or dare we say contract, with a specific *implementation* which is based entirely on the message schema honored by that implementation. The AddCustomer() method can be extended over time to handle multiple versions of schemas (with their own implementations) as well as other schemas appropriate to the business task of adding a customer.

While we could view this as a contract relationship between client and implementation, I think it is much healthier to stay with the covenant relationship at the service method level. It is best if the client doesn't realize that the method being called is actually a message router. That is black-box behavior that should be encapsulated by the service.

What About a Super-Router?

The next idea someone will toss out is that this means we can have one big message router. We don't need "services" and "methods", we can just have a super-router. And technically that is probably possible. Not desirable, but possible.

One school of thought in the OO world is that objects are defined by behavior. That an object should contain only behaviors appropriate to modeling a specific domain entity. This concept is valuable because it helps make objects intuitively comprehensible and somewhat self-documenting. It also provides relatively natural boundaries around objects, enabling concurrent development and other benefits.

The same benefits apply in the SO world. A service is a collection of procedures. It should be a collection of related procedures. Those procedures should be behavioral. In other words, each procedure should represent a specific domain behavior. Our covenants should be expressed in terms of those domain behaviors.

In other words, the service methods should be implementations of covenants. And those covenants should be expressions of the business tasks and processes we're simulating within the context of our computer system.

Looking to the Future

So does this all mean that we must give up Intellisense, compile-time checking, easier debugging and all the other benefits of a contract-based world? Not necessarily, but unfortunately the vendor community isn't universally enabling the scenario I've laid out here. The current tools are focused on methods having a single contract, both at the API and schema level.

What we need is the ability for a single method to have multiple schemas as part of its "contract". Given that capability, a client development tool could allow the developer to choose which of the schemas to use. The client would become bound to that schema, which is fine, because they are merely conforming to a specific covenant. The server is merely saying "you send me schema A, and I'll do B".

It is frustrating, because the tools are *so close*. And yet they are so far, and we are left in this unfortunate place where we're told to use message-based designs and yet are given tools that guide

us toward building component-based models.

I have suggested that Indigo (Microsoft's next generation Web services/Remoting/Enterprise Services technology) should directly support the Message Router concept for message endpoints. It is an idea they are giving serious consideration, so we may see support for this model directly enabled in Indigo.

If such a thing happens, we could very likely have the ability to create multiple implementations of a given method, each one accepting a different message schema. Inbound messages would be automatically routed to the correct implementation based on schema type, and messages with unrecognized schemas would be rejected. This idea is very comparable to method overloading in the OO world, but applied to services instead.

In the meantime, it is up to us to implement our message endpoints (service methods) as message routers, examining the schema of any received message and routing it to an appropriate implementation method. Even though this is a bit of extra work on our part, it is well worth the investment since it goes a long way toward solving not only the versioning issue but the larger issues of software evolution as well.

Rockford Lhotka is the author of numerous books, including the Expert One-on-One Visual Basic .NET & Expert C# Business Objects books. He is a Microsoft Software Legend, Regional Director, MVP and INETA speaker. Rockford speaks at many conferences and user groups around the world and is a columnist for MSDN Online. Rockford is the Principal Technology Evangelist for Magenic Technologies (www.magenic.com), one of the nation's premiere Microsoft Gold Certified Partners dedicated to solving today's most challenging business problems using 100% Microsoft tools and technology. For more information go to www.lhotka.net.

Authors

Rockford Lhotka is the Principal Technology Evangelist for Magenic Technologies, a company focused on delivering business value through applied technology and one of the nation's premiere Microsoft Gold Certified Partners. Rockford is the author of several books, including 'Expert One-on-One Visual Basic .NET Business Objects' and 'Expert C# Business Objects'. He is a Microsoft Software Legend, Regional Director, MVP and INETA speaker. He is a columnist for MSDN Online and contributing author for Visual Studio Magazine, and he regularly presents at major conferences around the world - including Microsoft PDC, Tech Ed, VS Live! and VS Connections. Rockford has worked on many projects in various roles, including software architecture, design and development, network administration and project management. Over his career he has designed and helped to create systems for bio-medical manufacturing, agriculture, point of sale, credit card fraud tracking, general retail, construction and healthcare.