



Part 1: What Is Service Oriented Architecture?

It sometimes seems like the software industry is always on the verge of a revolution--whether it is the promise of objects and object-orientation over procedural design, the arrival of components and COM, or a new language and environment like Java. The latest wave sweeping across the technology landscape is Web services and service-oriented architecture (SOA). Is SOA just another over-hyped panacea that will fail to deliver, or will it forever change the way software is constructed? The answer, we will see, is a little bit of both. We will take an introductory look at what SOA is, how it differs from the way software has traditionally been designed, and its implications for developers and managers considering a move to SOA.

On the Edge

Service-Orientation is fundamentally about the way software integrates with other software—how it behaves on the edges where it touches other software, if you will. Until recently, the primary way that good software made itself available to other software was through componentization. A component would present an interface to the outside world. This interface described methods and parameters. If we were in the Microsoft world, this interface would almost certainly be described in IDL, the description language for COM components. The interface was a contract between our component and the world. The world would program to our interface and, in return, we would always and forever honor the calls to it. This was a giant leap from the proprietary integration and reuse techniques we had previously. It was now very easy to take a component from someone that we have never met and plug it right into our application. The Visual Basic language adopted COM and made the IDL invisible to the programmer. Exposing any of our interfaces to the outside world was now so easy—too

"Service-Orientation is fundamentally about the way software integrates with other software—how it behaves on the edges where it touches other software, if you will."

easy in fact! Before long we had systems where every single interface was a public contract. Versioning became difficult to manage, to put it mildly. Systems were tightly coupled and became brittle and difficult to change. While the component revolution advanced the state of the art of the software industry and made software developers much more productive, there was a growing sense that we had not yet arrived at nirvana.

All About Abstractions

More than anything else, software design, or any design for that matter, is about abstractions. When the first brave souls programmed the first computers, they did it directly to the machine--about as concrete as you can get! Then someone had the idea of developing a mnemonic set of codes that represented the instructions and they let the computer translate the symbols into machine-readable code, and the first assembler was born. They probably wondered if life could get any better than that. Well, we all know the story. High level languages like C abstracted out concepts like looping, variables, etc. The next big jump in level of abstraction was object orientation. OO allowed a whole new level of abstraction, and new design patterns emerged to make software developers even more productive.

Interfaces between software applications rely on abstractions as well. And just as we have evolved ever more refined abstraction within our programming languages, we have developed ever more abstracted interfaces. The first method for plugging our software into someone else's software on the PC was to hook our software into just the right interrupt. We then developed the more civilized methods of public entry points into a DLL. Every time we wanted to program to a third party application, we had to not only learn the interface, but we had to make sure that we used the proper calling convention (Do I clean up the stack or the other guy? and other fun stuff). Then, with rise of Visual Basic, came the component age. As we discussed above, this was a colossal improvement in the way software developed by different groups at different times could be cobbled together to create complex and powerful applications in much less time. Component technology allowed us to abstract out the object and talk directly to it. We no longer needed to be concerned if it was coded in C++, VB, or any other language—all of those differences were abstracted away for us by the standardization of the particular component technology we chose. While this was a big step forward, we were still connecting our components at a binary level. With COM, we could only communicate with other components on other Intel/Windows platforms. We had to create a common type system for each component technology. The COM type system was only ever comfortable for VB programmers (What C++ programmer will miss the BSTR?!?). Because we were exposing objects, we had major problems with object lifetimes and reference counting.

Possibly even worse, as we have discussed, the outside world was writing code that was completely dependent on our class structures. While some very forward thinking people wrote their COM interfaces at a higher level of abstraction than their internal classes, most of us simply exposed our public methods through IDL. The siren song of the Microsoft tools making it so easy and the promise of actually shipping on time made it too hard to resist. Before long, we had a versioning nightmare. We needed a higher level of abstraction for the edges of our application where we interact with other systems.

"...instead of simply exposing our methods to the outside world, and expecting it to pass us types that we expect, we expose a service that expects a message in XML."

Enter Web Services

The rise of the World Wide Web (WWW) was a phenomenon that was less a creation of the software industry, and more of something that the industry reacted to, learned from, and eventually adapted to. The web was a model of heterogeneous, loosely coupled systems. The Hypertext Transport Protocol (HTTP) and HTML made it so that computers running on completely different platforms in

different time zones could interact as if they were on a local network. The only problem was that the HTML primarily connected human beings in front of browsers to systems. It wasn't well-suited to machine-to-machine communications. Once the first person had a web server consume and return XML instead of HTML, the Web service was born. Now any machine could communicate with any other machine across the network, as long as they both spoke XML over HTTP. The innovators of the industry saw the potential here and began to work on standardization. SOAP quickly emerged as the Web service standard. This standard had a little bit of a rough start, but has settled into a quite capable and accepted standard. In addition, the folks at IBM, Microsoft, and others have been building on the SOAP standard to provide protocols for making web services even more powerful and interoperable with standards like WS-Security, WS-Routing, WS-Transaction, and a handful of others.

But What Now?

Let's clear one thing up--web services are a great enabling technology, but they do not give us an architecture. Said another way... SOA is much bigger than web services. In fact, the first stumbling steps in the development of SOAP were an attempt at simply replacing Distributed COM (DCOM), and SOAP even stood for Simple Object Access Protocol. This was going to be how we accessed objects across HTTP. With a little time and thought, we realized that it was time for that next level of abstraction in software communication, and web services happened to be a perfect fit. Web services are naturally suited to pass messages to services instead of parameters to object methods. This is the sine qua non of SOA—instead of simply exposing

our objects' methods to the outside world, and expecting it to instantiate them and call methods on them, we expose a service that exists for the sole purpose of integration that expects a message in XML. We perform the service and optionally return a message (in XML of course) back to the requestor. The party using our software does not instantiate our objects and therefore has no interest in knowing what platform we are running on. The communication is over HTTP, so our requestor doesn't need anything but a network addressable endpoint (URL). The message that we get is in XML, where the X stands for eXtensible, so we are not locked into a fixed, binary structure in a particular type system.

This has solved many of the problems we had in the component world, but it opens up all kind of other questions. Where does this leave object orientation, is it now irrelevant? How do we do transactions across service calls? How do I design my new services interface? Does it matter what platform I build my services on? These are all questions that we will look into in depth in forthcoming articles.

About The Author

As a fifteen year veteran of the software industry, Eric has a proven track record of developing reliable and maintainable software systems that perform and scale. Learn more about Eric and how his skills and experience can bring value to your team and/or company at <http://www.sdiconsultinginc.com>.