



Newsletters



Classes



ObjectWatch



Speaking



Links

ObjectWatch Newsletter Number 45

October 7, 2003

This Issue: What is a Service-Oriented Architecture (SOA)?

CONTENTS

- ◆ Subscription Information
 - ◆ Quotation of the Month: The Lonely Architect
 - ◆ Main Article: What is a Service-Oriented Architecture?
 - ◆ CIPS Interview with Roger Sessions
 - ◆ Letters from Very Important People
 - ◆ ObjectWatch Services
 - ◆ Legal Notices
-

SUBSCRIPTION INFORMATION

Feel free to forward this newsletter, but don't make any changes. Thanks. There are now over 18,000 subscribers to this newsletter and over 70,000 readers! To find out about sponsoring an ObjectWatch Newsletter, contact [janet at objectwatch.com](mailto:janet@objectwatch.com).

For your own free email subscription, send mail to sub@objectwatch.com saying:

subscribe your-name, your-email

For example,

subscribe William Shakespeare, william@objectwatch.com

To unsubscribe, send mail to sub@objectwatch.com saying

unsubscribe your-name, your e-mail

QUOTATION OF THE MONTH: THE LONELY ARCHITECT

"Today, Web designers are called programmers, programmers are called engineers, engineers are called architects, and architects never get called."

- The Software Practitioner Triad by Alan Cooper in Software Architect, 9/14/03 available at http://www.ftponline.com/vsm/2003_09_14th/magazine/departments/softwarearchitect/

The winner of the Quotation of the Month Contest is Scott Bellware who gets a choice of a personally autographed copy of Roger Sessions's just released book, "Software Fortresses; Modeling Enterprise Architectures", or the ObjectWatch BBQ apron. Send in your nominations for Quotation of the Month to roger at objectwatch.com, and you too can get a personally autographed book/apron!

MAIN ARTICLE

WHAT IS A SERVICE-ORIENTED ARCHITECTURE?

I have been working with Stephen Fulcher of developerLabs to create a class called "Enterprise Process Integration with .NET". Our goal is to show how a large company, such as a bank or an insurance company, can architect and implement a cross enterprise heterogeneous integration strategy with .NET technologies and service-oriented architecture (SOA) principles.

As part of this preparation, I have been researching what various companies have to say about SOAs. As a result of this research, I have made two observations.

Observation one: Everybody loves SOAs. This much should be obvious. You can't read a recent article in any computer magazine without reading about SOAs. Every major software company is betting heavily on SOAs and trying to convince their customers to do likewise. If there is one thing on which the entire computer industry agrees, it is this: SOAs are IN!

Observation two: There is widespread disagreement over what an SOA actually is. Even worse, what little agreement there is on what an SOA is seems to me to be way off target. This is all rather alarming, given that SOAs are supposed to be the answer to interoperability. How can we use SOAs to achieve interoperability if we can't even come to a reasonable agreement on what an SOA is in the first place?

Most authors define an SOA in one of these three ways:

1. An SOA is a collection of components that can receive method requests over the Internet.
2. An SOA is essentially the next release of CORBA.
3. An SOA is an architecture for publishing and finding services.

All of these definitions are flawed. Let me go through them one by one. Then I'll give you a definition of an SOA that I think is more useful.

1. AN SOA IS A COLLECTION OF COMPONENTS RECEIVING METHOD REQUESTS OVER THE INTERNET

Many authors believe that SOAs are just collections of Internet-enabled components. For example, Kay Keppler, writing in JavaPro, says, "In an SOA, an application is built by assembling 'services,' or software components, that define reusable business functions. By using an XML interface between components, developers can construct

applications incrementally and reuse components.”¹

Equating SOAs with components is problematic because the term “components” is itself poorly defined. In the context of relating components to SOAs, the most reasonable definition of a “component” is a unit of distributed middle-tier-like functionality. Such a component would be built with managed components (until recently known as COM+) in the Microsoft world or a stateless session bean (part of Enterprise JavaBeans) in the Java world.

I have given a technical overview of such middle-tier components in other places². One distinguishing technical feature of these components is that each component-level method represents a subsumable transactional boundary. This is in contrast to object-level methods that almost never represent a transaction boundary, subsumable or otherwise.

By “subsumable” I mean that the logical transaction defined by the component method can, depending on run-time context, be subsumed into a higher level transaction.

Let me give you an example of subsumability in practice. Say we have three components, SavingsAccount, CheckingAccount, and AccountManagement supporting methods Withdraw, Deposit, and Transfer respectively. When we use the Withdraw method on SavingAccount to take out money, we want that withdrawal to be transactionally protected. Thus it should be a self-contained transaction. However when the Withdraw method is being called by the Transfer method, we want the withdrawal transaction to be subsumed into that of the transfer. So the Withdraw method must be both a transaction boundary AND subsumable, so that regardless of how it is invoked, it works correctly.

If transaction B (say, Withdraw) is to be potentially subsumed into A (say, Transfer), then it must be possible, at least under some circumstances, to have a transaction that spans both A and B.

If B includes some event, say B', that updates a database and A includes some event, say A', that updates some other database, then the fact that a transaction spans both A and B implies that at least some part of A' database (s) must remain locked until B' finishes.

The lock dependency of A' on B' implies that the time delay between the start of A' and the completion of B' must be very short. If it isn't very short, then A' will become an unacceptable bottleneck on the system.

There is no standard definition for what is “very short”, but there is one thing of which we can be positive. The time it takes to make a request over the Internet is much, much longer than “very short”.

Given all of this, you can see that a component that accepts requests over the Internet is a contradiction in terms. An SOA could be defined as a component. Or an SOA could be defined as something that accepts requests over the Internet. An SOA can't be defined as both. The transactional requirements won't support it.

2. AN SOA IS ESSENTIALLY THE NEXT RELEASE OF CORBA

Some authors believe that SOAs are the next release of CORBA. One such author is Judy Hunt of IONA (one of the CORBA leaders). Hunt writes, “The first widely adopted, standards-based SOA technology was CORBA³.”

Similar views are expressed by Bill Ruh, of ZDNet News who writes, “Technology pundits like to talk about SOA as if it were a new idea. But seasoned technology practitioners know better. SOA is really the latest stage in a gradual evolution of ideas that began in the early 1980s within the Object Oriented Programming community, and continued into the 1990s in other communities such as DCE, COM/DCOM, CORBA and J2EE⁴.”

CORBA stands for Common Object Request Broker Architecture. CORBA was an architecture of the early 90s and was highly influential in its time. CORBA was owned by a consortium known as the Object Management Group (OMG).

During the CORBA years, I worked at IBM. I was one of the lead architects of the CORBA persistence specification⁵ and the lead architect for IBM's first implementation of that standard. I am therefore very familiar

with the CORBA effort and very interested in any possible relationship between CORBA and SOAs.

CORBA had widespread support of most of today's key players of SOAs with the notable exception of Microsoft. Dozens of companies invested collectively hundreds of millions of dollars in CORBA technologies. The industry-wide interest in CORBA was unprecedented.

Although many authors point to the similarity between CORBA and SOAs as a good thing, I feel just the opposite. Why? Because despite tremendous early interest and large investments in CORBA, CORBA was ultimately a failure. Prudent companies that are considering investing in SOAs should therefore convince themselves that the failure of CORBA does not augur a similar fate for SOAs. The widespread proclivity to define SOAs in relationship to CORBA should, at the very least, raise an alarm.

In reality, however, I don't believe that CORBA's demise bodes poorly for SOAs. The reason is simple. Despite the widespread conjecture on possible family resemblances, SOAs and CORBA have nothing whatsoever to do with each other.

CORBA was a massive standardization effort. The CORBA activity fell into four main categories. The categories were as follows:

- ◆ APIs for various CORBA services including object persistence, the one in which I was involved.
- ◆ Definitions for language bindings.
- ◆ Standards for defining interfaces on components (or "objects", as they were known in CORBA).
- ◆ A standard for defining synchronous communications requests between components.

The first three categories collectively represented at least 95% of the CORBA effort. This large bulk was all in the area of portability. Portability standards have historically been encumbered with a severe handicap: vendors claim to follow the standards while at the same time peppering the standard with their own proprietary extensions. Within a short time "the standard" has so many proprietary enhancements that it no longer has value as an industry standard. This is exactly what happen to CORBA.

Even the 5% of CORBA that dealt with communications, as opposed to portability, had a fatal flaw. The standard that defined how components communicate was known as IIOP. IIOP defined both the nature of messages passed between components and the nature of the channels used to transport those messages. The IIOP definition for messages was reasonable. The IIOP definitions for transport, however, were not.

IIOP focused on synchronous communications channels. We have since learned that synchronous communications is a particularly poor way for systems to communicate.

SOAs, as I will discuss shortly, have nothing to do with either portability or synchronous communications. Therefore the problems that beset CORBA are not likely to impact SOAs. SOAs may end up with their own problems, but at least they don't have to worry about inheriting any from CORBA.

This differentiation between SOAs and CORBA is good news for the SOA community. And perhaps not very good news for the many companies that are trying to convince us that their experience with CORBA makes them natural experts in SOAs.

3. AN SOA IS AN ARCHITECTURE FOR PUBLISHING AND FINDING SERVICES.

The final misconception I will address is that an SOA is an architecture for publishing and finding services.

This view is taken by Rational/IBM, for example, who say, "So what is a service-oriented architecture (SOA)? In essence, it is a way of designing a software system to provide services to either end user applications or other services through published and discoverable interfaces." ⁶

Dan Gisolfi of IBM expands on this, writing "SOA is a conceptual architecture for implementing dynamic e-business." ⁷ He then discusses his view on the three SOA participants. They are the "service provider" that

represents some type of software asset. The service provider publishes its willingness to do specific work with a “service broker” that acts as “a repository, yellow pages, or clearing house”. This look-up facility is used by a “service requester” to invoke requests. This overall model can be described as the Publish/Find/Bind model for SOA.

In the Web Services world, Publish/Find/Bind maps to the standards defined by UDDI, WSDL, and SOAP. Since these are the only viable standards for Publish/Find/Bind in the world today, we can say that for all practical purposes, IBM views an SOA as anything that uses UDDI, WSDL, and SOAP.

However UDDI, WSDL, and SOAP are just technologies (or, more specifically, standards for technologies). Defining an “architecture” as anything that uses a particular technology is not very meaningful. It is like defining a house as anything that uses bricks. From a technology perspective, it is like defining an object-oriented architecture as anything that uses C++.

Equating the term “architecture” with a set of technologies is not only not meaningful, but a dangerous deception. The term “architecture” lends an aura of respectability. Things that are “architected” have a form, a logic, a coherence. I can assure you that there are many systems built using UDDI, WSDL, and SOAP that have neither form, logic, nor coherence. Companies that believe that the use of web service technologies implies an “architecture” in any meaningful sense of the world are deluding themselves. The systems they are building have about as much chance of working as a random pile of bricks has of spontaneously transforming itself into a house.

SO WHAT IS AN SOA?

So if an SOA is not just an Internet-enabled component and not just CORBA with a facelift and not just anything that happens to publish an interface with some UDDI implementation, then what is it?

Here is my starting definition of a Service-Oriented Architecture. A Service-Oriented Architecture (SOA) is a set of architectural principles that define how autonomous systems interoperate.

There are two key words in this definition: autonomous and interoperate. Both of these are important in the overall definition of an SOA. Let’s consider each.

“Autonomous” has these definitions in the dictionary: politically independent, self-governing; a free and independent moral agent; acting as an independent, self-regulating organism.

Software systems that are autonomous, then, have these characteristics:

- ◆ They are CREATED independently of each other.
- ◆ They are BUILT by well-defined groups.
- ◆ They WORK independently of outside systems.
- ◆ They provide SELF CONTAINED functionality. The functionality they provide would be useful even if it was not associated with any higher-level systems.

The second key word is “interoperate”, which means to cooperate, to work together, to interact in a meaningful way.

Notice that autonomous and interoperate are opposite ideas. Autonomous implies a lone wolf, one who works without regard for others. Interoperate implies cooperation with others. An SOA, therefore, is a set of principles that specifically deals with the tension between two contrasting ideas: autonomy and interoperability.

Designing an SOA, then, is likened to herding cats; to convincing autonomous systems that they should, under certain well defined circumstances, interoperate, and to providing those systems with a blueprint for how to do so. The SOA is this blueprint. The SOA principles are the rules that govern the group behavior of those cats. The SOA Architect is the cat herder.

By applying these ideas to different software granularities, we can see what is potentially an SOA and what is

clearly not.

An Employee component is not autonomous. It has meaning only within the context of a larger system. Therefore it is not an SOA, even if it does accept method requests over the Internet.

A Human Resource Management system, unlike an Employee component, is autonomous. It may choose to publish its interfaces with a UDDI and accept requests over SOAP. But in the end it is a loner system. A system cannot interoperate with itself, regardless of how many UDDI registered interfaces it supports and how many times it registers those interfaces with a third party. It is not, therefore, an SOA.

A corporate blueprint that defines how the Human Resource Management system makes request of the Payroll system and how both work with a Financial Planner COULD be an SOA. If the overall plan follows SOA principles, then it is an SOA. If it doesn't, then it is merely a very large pile of IT spaghetti.

What, then, are these SOA principles? I'll derive the five most important ones from my original definition of an SOA: interoperating autonomous systems.

If two systems are autonomous, then they have no mutual dependencies. This implies that neither system interrupts its own work to wait for some other system to complete its task. This implies asynchronous communications. The first SOA principle is therefore asynchronous communications.

If two autonomous systems interoperate, then there must be a communications channel between the two that is independent of the technology used by either system. The second SOA principle is therefore heterogeneous communications channels.

If two systems are truly autonomous of each other, then there is a natural healthy mutual suspicion that must be maintained. In order to get beyond this, there must be a way for each system to convince the other that it is really who it claims to be. The third SOA principle is therefore proof of identify.

It would be great if everything worked every time. Unfortunately, things don't always work. An architecture of interoperability must take into account what happens when things don't work as well as when things do work. The fourth SOA principle is therefore error management across the systems matrix.

When multiple autonomous systems are communicating with each other asynchronously, keeping track of where we are in the overall workflow effort is much more complicated than when systems are working together synchronously. The fifth SOA principle is therefore workflow coordination.

I can now expand the definition of an SOA to address these issues. My new definition of an SOA is as follows:

DEFINITION

An Service-Oriented Architecture (SOA) is an architecture that defines how autonomous systems interoperate with particular focus on:

- ◆ Asynchronous communications
- ◆ Heterogeneous transport channels
- ◆ Proof of identify
- ◆ Error management
- ◆ Workflow coordination

Notice that there is one similarity between this definition of SOAs and IBM's definition. We both agree that heterogeneous transport is critical. While IBM doesn't say this outright, this is the implication of their SOAP discussions.

Beyond this similarity, there is little overlap. For example, my definition does not place any particular importance on the Publish/Find capabilities. I believe that a relatively small and unimportant number of SOA interactions require Publish/Find. And the IBM definition does not include any of my five bullets beyond the SOAP/heterogeneous communications.

I hope this article moves the discussion on SOAs forward in a positive direction. It would certainly be helpful for those of us trying to teach how to use SOAs if we could have some general agreement on what an SOA actually is.

- Roger Sessions
October 7, 2003
Austin, Texas

(1) The Next Big Thing by Kay Keppler in JavaPro, August 2003, available at http://www.fawcette.com/javapro/2003_08/magazine/departments/ednote/

(2) See, for example, my book, "COM+ and the Battle for the Middle Tier" in which I deal with this topic extensively. It is published by John Wiley and Sons 2000, ISBN 0471317179.

(3) - A Brief History of Service-Oriented Architectures, Part 1. By Jody Hunt, IONA. Available at http://www.iona.com/emea/emea_news/0305_uk%20feature.pdf

(4) - Succeeding at Service Oriented Architectures by Bill Ruh, ZDNet News. Available at http://zdnet.com.com/2100-1107_2-5058101.html and featured at XLM.ORG.

(5) - For more information on CORBA persistence as well as my own role in the CORBA work, see my book, "Object Persistence; Beyond Object-Oriented Databases", published by Prentice-Hall. ISBN: 0131924362

(6) - Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications by Alan Brown, Simon Johnston, and Kevin Kelly available at <http://www.rational.com/media/whitepapers/TP032.pdf?SMSESSION=NO>

(7) Web services architect: Part 1 - An introduction to dynamic e-business by Dan Gisolfi available at <http://www-106.ibm.com/developerworks/webservices/library/ws-arc1/>

CIPS INTERVIEW WITH ROGER SESSIONS AVAILABLE

Stephen Ibaraki of The Canadian Information Processing Society recently conducted an interview with Roger Sessions. You can read that interview at <http://www.cips.ca/news/national/news.asp?aid=1698>

LETTERS FROM VERY IMPORTANT PEOPLE

We had several important responses to our last newsletter. Because of space constraints, we are going to send them out as a separate newsletter .

OBJECTWATCH SERVICES

Let ObjectWatch and Roger Sessions help your company be successful.

Here are some of the services we offer:

- ◆ Executive Briefings that summarize critical information in brief sessions.
- ◆ Classes and workshops for your architects and developers.
- ◆ Architectural roundtables.
- ◆ Training programs for developers.
- ◆ Keynotes for large conferences.
- ◆ Consulting on system architectures.
- ◆ White papers on topics related to large enterprise architectures.
- ◆ The best newsletter in the world for software architects (but of course, you already knew that!).

Our Areas of Expertise:

- ◆ The theory and practice of Service-Oriented Architectures
- ◆ The Software Fortress Model, and how to apply it in designing large SOA systems.
- ◆ Issues related to enterprise software systems.
- ◆ Emerging standards related to enterprise software systems.

Just write Janet (at [objectwatch.com](mailto:janet@objectwatch.com)) and let her put together a program that meets your specific needs.

Legal Notices

The ObjectWatch Newsletter does not rent out its subscription list.

ObjectWatch accepts one sponsoring advertisement per issue. To find out about sponsoring an ObjectWatch Newsletter, contact [janet at objectwatch.com](mailto:janet@objectwatch.com).

This newsletter is Copyright (c) 2003 by ObjectWatch, Inc., Austin, Texas. All rights are reserved, except that it may be freely redistributed provided that it is redistributed in its entirety, and that absolutely no changes are made in any way, including the removal of these legal notices.

ObjectWatch is a registered trademark (r) of ObjectWatch, Inc., Austin, Texas. Software Fortress is a trademark (tm) of ObjectWatch, Inc., Austin Texas. All other trademarks are owned by their respective companies.
