Presents

Simple benefits of Service-Oriented Architecture (SOA)

October 2004

Currently, the big architectural push is towards a service-oriented architecture (SOA). While the term sounds impressive, it is important to understand both what an SOA is and how it can benefit an enterprise. There are a number of different links available that describe the components and concepts behind SOA, including at ibm.com, xml.com, and microsoft.com. Given this wealth of information, it would be redundant to cover it here. Instead, I'd like to focus on some of the benefits that accrue to those who choose to follow the SOA path.

Even though I will not fully cover the concepts of SOA, it is still necessary that we have a common point of reference for a discussion of benefits. So briefly, a service-oriented architecture is a style of application design that includes a layer of services. A service, from this limited perspective, is just a block of functionality that can be requested by other parts of the system when it is needed. These services have a published as set of methods, known as an interface. Usually, each interface will be associated with a particular business or technical domain. So there could be an interface used for authentication and another interface used to create and modify a sales order. The details of the provided functionality are not important for this article so much as the relative location of the code that implements the functions.

A word of warning to developers who are new to the services architecture (or an object-oriented environment as well). SOA is as much about a mind set as it is about coding. In order to reap any of the benefits of SOA, the application must be designed with services in mind. It is difficult to convert a procedural application to services and still achieve anything remotely close to what can be gained from baking services in from the beginning. That having been said, there are usually elements of procedural applications which can be 'servicized'. This is the equivalent of refactoring the applications looking for common functionality.

And so on to the benefits. In each case, we are talking about development teams and enterprises that embrace SOA to the point that most of the development effort is focused on the creation and utilization of services.

Loosely Coupled Applications

Strictly speaking, coupling is the level of impact that two modules have on one another. From a developer's perspective, it is the odds that a change in one module will require a change in another. We have all seen tightly coupled systems. These are applications were developers are afraid to touch more procedures than necessary because they might cause the application to break. Whether we know it or not the goal is to create a loosely coupled environment. And services go a long way towards making such an environment a reality. Consider for a moment what I consider to be the epitome of loosely coupled systems: a home entertainment system (Figure 1).

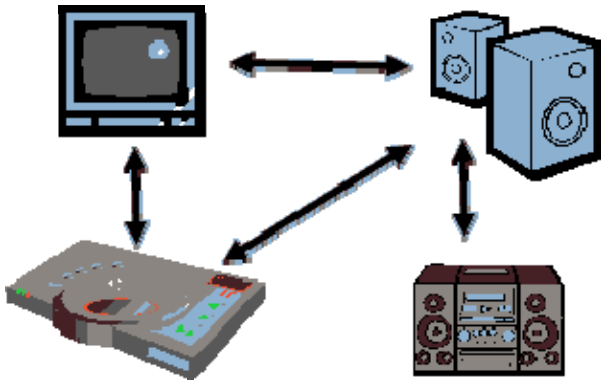

Figure 1 - A Loosely Coupled Home Entertainment System

Here are a whole bunch of different components (the exact number depends on how into electronic gadgets you are), frequently created by different manufacturers. While the user

interface can get complicated, between the devices the connections are quite simple. Just some wires that get plugged into standard sockets. Need to replace one of the element? No problem. Remove the wires connecting Component A to the system. Take Component A away. Insert Component B. Put the wires into the appropriate sockets. Turn things on and go. Talk about loosely coupled!

In software, a loosely coupled environment allows for changes to be made in how services are implemented without impacting other parts of the application. The only interaction between the application and the services is through the published interface. Which means that, so long as the interface doesn't change, how the functions are implemented are irrelevant to the calling application. Strict adherance to such an approach, an approach which is strongly enforced by SOA, can significantly reduce development by eliminating side-effect bugs caused by enhancements or bug fixes.

Location Transparency

Another benefit accrued through the use of interfaces and the loosely coupled structure imposed by SOA is location transparency. By the time we're done, it will seem like many of the benefits achieved with SOA will come from this same place (interfaces). In this instance, location transparency means that the consumer of the service doesn't care where the implementation of the service resides. The desired service could be on the same server, on a different server in the same subnet or even someplace across the Internet. From the caller's perspective, the location of the service has no impact on how a request for the service is made.

It might not be immediately obvious why location transparency is a good thing. And, in the most straightforward of environments, it doesn't really make a difference. However as the infrastructure supporting a set of Web services becomes more complicated, this transparency increases in value. Say the implementing server needs to be moved. Since the client doesn't care where the service is, the change can be made with no change required on the client. In fact, the client can even dynamically locate the implementation of the service that offers the best response time at the moment.

Code Reuse

Functional reuse has been one of the holy grails for developers since the first lines of code were written. Traditionally, however, it has been hard to achieve for a number of reasons. It is difficult for developers to find a list of the already implemented functions. Once found, there is no document that describes the expected list of parameters. If the function has been developed on a different platform or even in a different language, the need to translate the inbound and outbound values is daunting. All of these factors combine to make reuse more of a pipe dream than a reality.

The full-fledged implementation of a SOA can change this. The list of services can be discovered dynamically (using UDDI). The list of exposed methods, along with the required parameters and their types, are available through a WSDL document. The fact that even complex data structures can be recombined into a set of basic data types held together in an XML document makes the platform and language barrier obsolete. When designed and implemented properly, SOA provides the infrastructure that makes reuse possible. Not only possible, but because it is easy to consume services from Java (J2SE or J2EE), .NET or even the more traditional languages (C/C++, COBOL etc…), developers are much more likely to use it. And ultimately, the lack of developer acceptance has always been the biggest barrier of all for code reuse.

Focused Developer Roles

Almost by definition, a service-oriented architecture forces applications to have multiple layers. Each layer serves a particular purpose within the overall scope of the architecture. The basic layout of such an architecture can be seen in Figure 2.

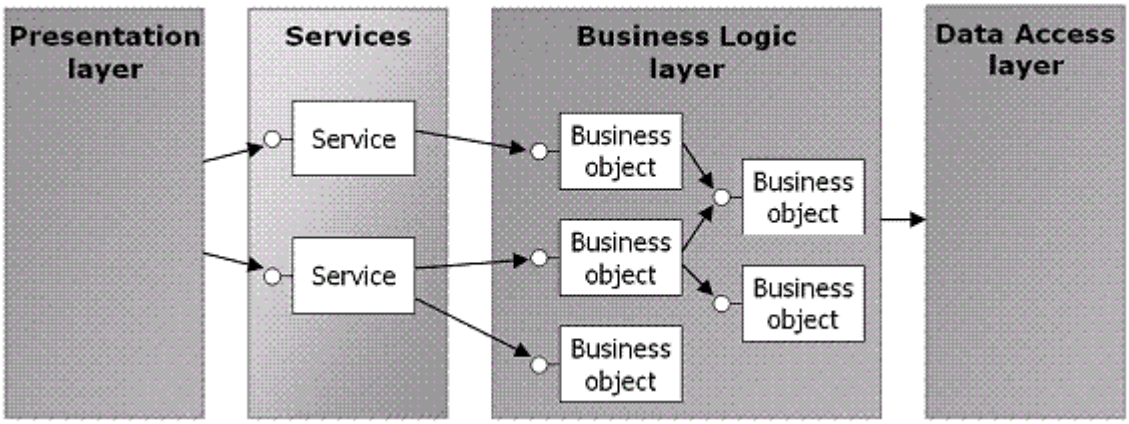

Figure 2 - Some Layers in a SOA

So when designing an application, the various components get place into one of the levels at design time. Once coding starts, the someone is actually going to create the component. The skills necessary to develop each of the components are going to differ and the type of skill required will depend on the layer in which the component is placed. The data access layer requires knowledge of data formats, SQL, JDBC or ADO.NET. The authentication layer could utilize LDAP, WS-Security, SHA or other similar technologies. I'm sure you get the idea enough to put it in play in your own environment.

The benefit of this separation is found in how the development staff can be assigned to the various tasks. Developers will be able to specialize in the technologies for a particular layer. As time goes by, they will be experts in the complex set of skills used at each level. Ultimately, by focusing the developers on a particular layer, it will even become possible to include less skilled programmers in a paired or a mentoring environment. This doesn't eliminate the need for an overall technical lead, but it certainly decreases the reliance of a company on experienced generalists.

Greater Testability

The fact that services are only accessedthrough their interfaces is the reason for this benefit. The fact that services have published interfaces greatly increases their testability.

Some of the nicest tools in recent years for assisting with the testing process are JUnit (Java) and NUnit (.NET version of JUnit). JUnit or NUnit allows for the creation of a test suite. The test suite consists of a number of procedures, each of which is designed to test one element of an application or service. JUnit/NUnit allows the suite to be run whenever required and keeps track of which procedures succeed and which fail. My experience with JUnit/NUnit is that it makes the process of developing a test suite incredibly simple. Simple enough that, even with my normal resistance for testing, I no longer had an excuse not to create and use unit tests.

The fact that services publish an interface makes them perfect for JUnit/NUnit. The test suite for a particular service knows exactly what methods are available to call. It is usually quite easy to determine what the return values should be for a particular set of parameters. This all leads to an ideal environment for the 'black box' testing that JUnit/NUnit was developed for. And whenever a

piece of software can be easily tested in such an environment, it inevitably results in a more stable and less bug prone application.

Parallel Development

There is another side benefit that arise both from the increased testability (using tools such as JUnit/NUnit) and the division of components into layers. Since the only interaction that exists between services is through the published interfaces, the various services can be developed in parallel. At a high level, so long as a service passes the JUnit/NUnit test for its interface, there is no reason why it would break when the various services are put together. As such, once the interfaces have been designed, development on each of the services can proceed independently. Naturally, there will need to be some time built into the end of the project plan to allow for integration testing (to ensure that the services can communicate properly and that there was no miscommunication during development). But especially for aggressive project timelines, parallel development can be quite a timesaver.

Better Scalability

This particular benefit refers to the ability for a service to improve its response time without impacting any of the calling applications. It is an offshoot of the location transparency provided by SOA. Since the calling application doesn't need to know where the service is implemented, it can easily be moved to a 'beefier' box as the demand requires. And if the service is offered through one of the standard communication protocols/middleware (such as HTTP or MQ Series), it is possible to spread the implementation of the service across a number of servers. This is what happens when the services is place in a web farm, for example. The reasons and techniques for choosing how to scale a service are beyond the scope of this paper. It is sufficient, at this point, to be aware that they are quite feasible.

Higher Availability

Location transparency also provides for greater levels of availability. Read the section on scalability for the fundamental reason. But again, the requirement to utilize only published interfaces brings about another significant benefit for SOA.

Building Multi-Service Applications

I describe this benefit as the “Star Trek Effect“. The idea can be seen in many Star Trek episodes, both new and old. In order to solve an impossible problem, the existing weapon systems, scanning systems and computer components need to be put together in a never-before seen configuration.

“Cap'n. If we changed the frequency of the flux capacitor so that it resonated with the dilithium crystals of that alien vessel, our phasers might just get through their shields. It's never been done before, but it might work. But it'll take me 10 seconds to set it up“

Try reading that with a Scottish accent and you'll get my drift. Here are people working with services to the level that no programming is required to be able to put them together into different configurations. And this is the goal that most service-oriented architects have in mind, either consciously or subconsciously, when they design systems. Although we are not yet at the Star Trek level of integration and standardization, SOA provides a good start.

Summary

And that's all SOA is, after all, a good start. But it is a start that is necessary to gain the benefits I've just run through. Sure it takes a little longer to design a service-based application. Not to mention the creation of the infrastructure that needs to surround it, assuming that it will be production quality. However in the long run, the benefits start to play out.

Now I'm not naive enough to believe that we haven't heard this song before. It is basically the same song that has been playing since object-oriented development was first conceived. We heard it with COM, DCOM, and CORBA. Now it's Web services and SOA. Why should we believe that this time will be different from the others?

I can't offer any reason in particular in answer to that very astute question. Honestly, SOA feels different than COM and OO. Perhaps that is because we've already seen the problems caused by COM, DCOM, OO, CORBA, etc and they are addressed by Web services. Perhaps it is because there is now a common language that can be used across platforms (XML). Perhaps it is the level of collaboration between the various vendors regarding service-based standards. Perhaps it is the fact that SOA accounts for legacy systems whereas the other paradigms demanded a costly and impractical re-write of legacy systems. But to me at least, all of these items make me much more optimistic that this time we are heading down the right path.