GETTY IMAGES

A

Classic Ex

Take a low-t

S

ample

ech approach
to understanding

OA

by Dan North

During the past few years, service-oriented architecture (SOA) has become a mainstream approach for designing and developing enterprise software. As with most new technologies, the adoption of SOA has been led by software vendors and shaped by their particular tools and sales targets. Naturally it is in the vendors' interest to emphasize the complexity of SOA and then provide a timely and profitable solution.

This leads many systems architects into a technology-centric view of SOA, when, in fact, the most important criteria for a service-oriented architect—before tackling the technology—should be a keen understanding of the business.

This article presents a simple, technology-agnostic approach to designing and evolving SOAs. You will not see acronyms such as WSDL, SOAP, or REST, and I promise not to use technical terms like “orchestration,” “realization,” and “governance.” Because of this, you will be able to design and implement service-oriented architectures that truly serve your business.

Describe a Scenario in Business Terms

Imagine you want to implement a vacation-booking service as part of an enterprise system. The first step is to remove any reference to computers or modern technology. This will allow you to concentrate on the business objectives of the service, without getting sidetracked by technological considerations. In other words, it enables you to separate the “what” from the “how.”

“...can we assure consumers what they can expect from our service? Can we guarantee it?”

A simple way to do this is to describe the business interactions as they would have occurred in a 1950s company. Let's call our company Big Corporation, Inc. It's a classic 1950s corporation—hierarchical, with myriad departments containing countless people who are cogs in the Big machine. They all know their places, their job functions are clearly defined, and there are no messy organizational boundaries. Remember, the job title “project manager” doesn't exist yet. You can use a telephone and the internal mail but nothing else. Here is how the scenario might go:

Bob Books a Vacation

Bob works for Big Corporation, Inc. Bob wants to take a vacation. He's been working hard, and he thinks he deserves it. So Bob goes to the stationery shelf, takes an annual leave request form, and diligently fills it in. It's quite a basic form—he fills in his name, his department, the dates he wants to take off, and then he signs the form. For good measure, he writes his telephone extension on the bottom of the form. He takes the form to his line manager, who countersigns it, and then he puts it in the internal mail. Oh, the internal mail! It's notoriously unreliable, but he sends off the form anyway and offers up a silent prayer. He puts a note on his desk calendar to remind him to follow up in a week.

Then he gets back to work on his important administrative job.

A week later, Bob's calendar reminds him that he hasn't heard back from the personnel department, so he decides to phone them.

Bob who? No, we don't have a leave request for you. Yes, of course I'd know—I handle all the leave requests. Sorry!

Undeterred, Bob fills out another request form, puts it in the internal mail, and gets back to work. A couple of days later an envelope drops into Bob's inbox. His vacation has been approved—great news!—now to book the hotel and tell the wife and kids.

Replay the Scenario in Service-oriented Terms

Now, let's replay this scenario and identify where services are being used and what the interactions look like.

In SOA terms, the personnel department is a service provider. One of its services is scheduling annual leave. Bob is a client or consumer of the service. The vacation request form describes a service contract. The filled-in form is then an asynchronous message—Bob doesn't suspend his activities waiting for the reply to come back; instead he gets on with his regular work.

The internal mail is a message transport—an unreliable one in this instance. Luckily, Bob has an error-handling protocol, namely a timeout. After a week, his calendar alerts him to call the personnel department. The telephone call is a synchronous interaction—he is on the telephone in real time. He then implements an error-correcting strategy, which is to resend the message. Notice that the whole sequence—the reminder on the calendar, the timeout, the telephone call, and resending the message—was only necessary because the message transport was unreliable. We'll return to this idea later.

A second asynchronous message is sent, and this time Bob receives a response in his inbox. This is an asynchronous response. Finally, Bob correlates the response with the original request, and the exchange is complete.

Use the Business-based Scenario to Explore Service-oriented Assumptions

Unless you are an SOA veteran, the technical terms in the previous section may be unfamiliar to you. However, they have a direct, one-to-one relationship with the terms in the business version, which has the advantage of being real and tangible.

Using the previous example, here are some discussions and questions that will help clarify the assumptions your architects might make about defining the service:

RELIABILITY OF MESSAGE TRANSPORT

We discover from the description that the message-delivery system—the internal mail—is unreliable. Is this something over which we have any control? Could we make it more reliable or even replace it altogether? If not, what are the constraints under which we are working, and what level of service can we expect?

ERROR-HANDLING STRATEGY

The scenario doesn't describe how the first message is lost. Was it never delivered to the service, or did it go missing after it arrived at the personnel department? In the latter case, there may be more error conditions to consider. For example, if someone updates Bob's records to reduce the number of leave days remaining but the request goes missing before the vacation has been approved, his records are in an inconsistent state. The records won't reflect reality—the personnel department has "lost" some of Bob's vacation time. In other words, the vacation-booking service has to be transactional. It is similar to the situation in which you debit one account and credit another—either both actions must succeed or both must be undone.

There is a clue in the telephone conversation. It seems there is only one person who deals with vacation requests, so it is less likely that she will lose the request partway through than if several people were involved in the process. This is a feature of the implementation of the service: The more complex it is, the more we need to consider transaction boundaries and the ramifications of failures in different parts of the process.

The errors we have discussed so far depend on how the request is performed (i.e., how it is implemented). But there are several business reasons why a vacation request might fail. In SOA terms these are known as semantics or business rules. What if Bob already has used up his allowance or this request would take him over his limit? What if the manager who signed the form isn't authorized to sign vacation requests? In these cases, we might want to send a response back to Bob explaining why we can't process his request.

"...a true service-oriented architecture should only have services with a direct counterpart in the business; otherwise they cannot possibly be modeling a business process."

CORRELATING RESPONSES

Bob's response arrives some time after he sent the original request, and he has been busy doing other things in the meantime, so he has to be able to correlate the response with his request. Of course, this is easy because he only has one outstanding vacation request. If he were sending and receiving a lot of the same types of requests, all asynchronously, he would need a more robust approach to correlating the responses as they come in, such as a reference number to match against.

Similarly, if there were many people sending in vacation requests at the same time, the internal mail system might need some help to avoid becoming overwhelmed and thus delivering responses to the wrong inboxes or losing them entirely.

It is useful to understand the expected usage patterns of the service in terms of peaks and valleys and the maximum amount of concurrent usage. This may vary by time of day, week, month, or year. It may increase or decrease at weekends or monthends, or it may be seasonal. In our example, vacations are cyclical, with winter and summer peaks. We should determine if the service is international, in which case winter and summer may occur at different times for different users. This may figure in our discussions around correlating the asynchronous responses.

AVAILABILITY OF SERVICE

Related to the idea of peak usage is the more general conversation concerning availability of service. Should we provide a service level agreement (SLA) that promises how quickly the service will respond under various circumstances? If not, can we assure consumers what they can expect from our service? Can we guarantee it?

Should we prioritize some services over others, and if so, how? For instance, if the personnel department also books business travel, it might process all business travel before any vacation bookings. Alternatively, it could guarantee to provide a certain percentage of its capacity to each type of request.

SECURITY AND COMPLIANCE

In this example, security probably is not an issue. It is unlikely that someone would want to fake a vacation request on Bob's behalf. However, what do we know about Bob himself? Perhaps his boss turned down his request, and Bob forged his manager's signature. This might be more likely if it were left to the manager to track Bob's vacation allowance rather than the personnel department. In this case, we might consider a more careful verification of the request, such as a (synchronous) telephone call to Bob's manager to ensure that she really had signed the request.

In addition to our own business rules, we must consider external legal constraints that may be imposed on us. This is particularly true in systems that maintain sensitive financial or personal information.

SOME OBSERVATIONS

As you can see, we have been able to ask some useful questions, which are helping us understand the nature of the service without descending into technological discussions or becoming distracted by implementation details. Keeping the discussion at a business level enables us to understand the true business value and intent of the service and allows us to investigate any business complexity independent of technical issues.

In fact, a true service-oriented architecture should only have services with a direct counterpart in the business; otherwise they cannot possibly be modeling a business process. Abstract technical concepts, such as a "data service" or a "replication service," don't make any sense in business terms. (There may be shared modules or libraries to provide common technical functionality, but these should not be implemented or referred to as services.)

A useful heuristic is to equate a service provider with a department or team within the organization. For instance, we might model the personnel department as a single service provider offering a number of services, or we might partition it into smaller service providers if the services naturally fall into categories.

Evolve the Service in Business Terms

So far we have only talked about the initial design of a service-oriented architecture, but we also want to be able to change and adapt services over time. To understand the issues involved here, it is useful to look at why a service might evolve. Again, we start with a business-focused scenario and then map it into SOA terminology.

Susie Saves the Day

Susie works in the personnel department of Big Corporation, Inc. processing the vacation requests. One day she receives a vacation request from Bob, so she looks up his remaining leave allowance and discovers he doesn't have enough days left. She notices that the vacation is imminent—he wants to leave in two days. By the time the response gets back to him and he resubmits the request, it will be too late. She notices that he has scribbled his telephone extension on the bottom of the form, so she calls him.

Hey Bob, Susie from personnel here. I'm afraid I have to turn down your request for three weeks' vacation; you don't have enough days left.

Three weeks? I only meant to book two weeks! What dates do you have on that form?

The 1st through the 21st.

Oh gee, sorry, that should say 1st through 14th. I must have been distracted. I'm reading this article about the flying cars we'll all have in the year 2000.

Never mind. I'll approve this; you enjoy your vacation.

Thanks, Susie. You really saved the day!

Over time she notices more and more people writing their phone numbers on the form, so she is able to resolve problems more quickly.

She decides to add a new, optional field on the form for the extension number and calls the form "Vacation Request Form version 2."

Some time later, she receives a memo about a new management directive. Apparently all vacation requests now have to contain the name of the person who will be in charge in the vacationer's absence. No exceptions, those are the rules, effective immediately.

So she creates a new form, version 3, and sends out an all-company memo saying that, regretfully, versions 1 and 2 will no longer be accepted. She also drafts a standard response to send to anyone who submits any of the old-style forms.

Replay the Scenario in Service-oriented Terms

Replaying this scenario, we see the service evolve twice, and for different reasons. In the first instance, we realize we can provide an enhanced service if the request has some additional data. We publish a second version of the service contract—the blank form with the optional field added.

Notice that we didn't withdraw all the version 1 forms. We still can provide a service to clients using the old version of the service contract. All that happens is that we gracefully degrade the service because we can't call the requester if there are any problems. This allows clients to choose when to upgrade to the new service contract and get the new features and provides loose coupling between upgrading the service and the clients.

This is the most common form of service evolution. Either through feedback from your users or ideas from your own organization or your competitors, you will want to add new functionality or improvements to your services.

BE RESPONSIBLE WHEN MAKING A CHANGE THAT BREAKS EXISTING CLIENTS

The second situation is somewhat different. A new, mandatory piece of data is required, without which the service cannot be implemented. This means that no consumers can use the service with their existing service contract and everyone has to "upgrade" to the new form. Because we are responsible service designers, we provide an error response rather than silently discarding any outdated requests. This strategy of providing immediate feedback to the consumer is known as "failing fast."

It is useful because the client now can see exactly why the request failed (and will continue failing until he upgrades his request format). Otherwise he could assume the failure was due to some infrastructure issue or temporary service outage and just keep trying.

However carefully you try to evolve a service, a sudden step change can occur, especially in heavily regulated environments where the regulators are outside of your control.

AVOID A UNIVERSAL DOMAIN MODEL

Before we leave Big Corporation, Inc., there is one more analogy we can draw from the scenarios. It involves understanding the models that Bob and Susie have in their heads. When Bob thinks about a vacation, he might be imagining palm trees, activities for the kids, long walks on the beach—vacation stuff. When Susie thinks about a vacation, she thinks about updating files, filling in forms, and contacting line managers.

When they are talking to each other, they use the common word "vacation" but it means two totally different things. In the former case, it refers to a period of time away from the office; in the latter case it refers to the business process around booking that time.

In technology terms, each of them has a domain model of a vacation. Increasingly we are seeing organizations attempt to introduce "enterprise information architecture," "universal data dictionary," or some other fancy term for trying to force everyone to use the same domain model.

In our example, it would be not only difficult but also detrimental for Bob to share Susie's domain model of a vacation. As the personnel department grows, we might want to re-engineer the business process behind the vacation-booking service (for instance, by scaling up Susie's role to be performed by several people) which would be harder to achieve if it were coupled to Bob's (and everyone else's) understanding of a vacation.

Instead, we introduce the idea of a business concept. This is effectively the higher-level, ubiquitous language that ties together all of the finer-grained domain models behind each service. So "vacation" becomes a business concept that is interpreted differently by Bob and Susie. They each have a domain model that maps onto a vacation, but in each case, it is specifically tailored for their own purposes.

The service contract is then expressed in terms of enterprise-level business concepts, such as a vacation or a dispatch or a sales order, which again decouples the service consumer from the service provider and allows them to evolve independently, while still able to communicate in a common language.

The mistake that enterprise information architects (or people with similarly named roles) make is trying to define what the business concept means to each of the people using it.

Summary

The metaphor of a 1950s corporation allows us to define and discuss complex business interactions in terms of a service-oriented architecture, while protecting us from the issues of implementation details, tools, and products. By keeping the discussion focused on real business scenarios and away from technology, the business is able to play a useful and important role in identifying the requirements of an SOA.

Clearly, this is only the first step to implementing a fully working SOA for the problem at hand, and immediately following this exercise you would dive into the details of how to implement a solution to the specific problem. What it means, though, is that at any point the technical decisions can be mapped back to identifiable business value in terms of the business process we are trying to model. {end}

Dan North is a senior consultant with ThoughtWorks, where he coaches development teams in agile software delivery and project automation. A programmer with fifteen years of delivery experience, he has published a number of articles and spoken at conferences on topics ranging from agile enablement to NLP. Dan thanks Arjen Poutsma, Mats Helander, and his colleagues at ThoughtWorks for their help with this article. You can contact Dan at dan.north@thoughtworks.com.

DO:

1. HAVE A USER (!)
   - **Embedded in the business (he should not be a technical person)**
   - **Who cares about the outcome**
2. PASS AROUND FORMS AND DOCUMENTS, NOT OBJECTS (OR REPRESENTATIONS OF OBJECTS)
3. REMEMBER THAT CALLING ACROSS THE NETWORK TAKES TIME
4. USE COARSE-GRAINED SERVICES RATHER THAN A LOT OF LITTLE CALLS

DON'T:

1. DESIGN FOR WHAT YOU DON'T NEED
   - **Does it really need to be available 99.999% of the time?**
   - **This includes security, availability—in fact all the "-ilities"**
2. "PHONE HOME," I.E., DON'T MAKE SERVICE CALLS TO THINGS THAT ARE AVAILABLE LOCALLY
   - **Better still, don't expose them as services in the first place**
3. CREATE A SERVICE IF YOU ONLY HAVE ONE CLIENT
4. EXPOSE YOUR PRIVATES!
   - **In other words, avoid putting implementation details into the message**
5. HAVE TRANSACTIONS ACROSS MULTIPLE SERVICE CALLS
   - **Instead package all the calls into a single, coarse-grained service call**