Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 3. Service realization

Skill Level: Introductory

Jim Amsden (jamsden@us.ibm.com)
Senior Technical Staff Member
IBM

21 Jan 2010

This third article of this five-part series explains how services are actually implemented. The service realization starts with deciding which participant will provide and use what services. That decision has important implications in service availability, distribution, security, transaction scopes, and coupling. After these decisions have been made, you can model how each service functional capability is implemented and how the required services are actually used. Then you can use the UML-to-SOA transformation feature included in IBM® Rational® Software Architect to create a Web services implementation that can be used in IBM® WebSphere® Integration Developer to implement, test, and deploy the completed solution.

About this series

In the first article in this series, "Part 1. Service identification," (see "View more content in this series"), we outlined an approach based on the IBM® Service-Oriented Modeling and Architecture (SOMA) method for identifying services that are connected to business requirements. We started by capturing the business goals and objectives necessary to realize the business mission. We then modeled the business operations and processes that are necessary to meet the goals and objectives. Then we used the business process to identify capabilities that are candidates for exposure through service interfaces.

In the second article, "Part 2. Service specification," we modeled the details of the service interfaces. A service interface defines everything that potential consumers of

the service need to know to decide whether they are interested in using the service, plus exactly how to use it. It also specifies everything that a service provider must know to successfully implement the service. That is, a **service interface** defines the possible interactions of consumers and providers through specific interaction points.

In this article, we'll look at designing how the services are actually provided or, in Unified Modeling Language (UML) terminology, *realized*. The service realization design starts with deciding which participants will provide and use what services. That decision has important implications in service availability, distribution, security, transaction scopes, and coupling. After these decisions have been made, you can design how the functional capability of each service is to be implemented and, therefore, how the required services are actually used. Note that we are not constrained to a particular level of abstraction in designing our services implementations. We could be implementing services across different corporations, services of participants in a manual business process, or information technology services in some software platform. The same concepts apply at any of these levels of abstraction or across different concerns. The point is to partition services and requests among loosely coupled participants assembled in a service value chain.

The next article in this series, "Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 4. Service composition," will describe how these services can be composed to create new services. The final article, "Part 5. Service implementation," will use the IBM® Rational® Software Architect UML to SOA transformation feature to create a Web services implementation that can be directly used in IBM® WebSphere® Integration Developer software to implement, test, and deploy the completed solution.

Context of this article

A complete understanding of SOA modeling requires getting to the details of how a service is actually implemented by providers and used by consumers. If the implementation is difficult, then perhaps the specification is incorrect or the wrong services have been identified. This article shows how to design the implementation of each of the service interfaces that we developed in the previous article. The implementation design consists of three steps:

1. Decide which service providers provide which services.
2. Design the service implementations.
3. Assemble and connect service consumers and providers that are necessary to model complete implementations.

Deciding which services are provided by which providers (there could be more than

one) is driven by many factors, including:

- Functional cohesion to maximize reuse
- Where the services are most likely used
- Where they are most likely to be deployed
- What qualities of service are required
- Stability of the functional area
- Where the most change is anticipated
- How much coupling is tolerable in the domain
- Reducing coupling to minimize the effect of change
- Security issues
- Applicable platform implementation technologies
- Integration and reuse of existing systems

A detailed analysis of all of these concerns is beyond the scope of this article, but it is covered fully in the IBM® SOMA method. Here, we'll assume that, somehow, the IT architect has decided which participants will provide what services, so we can focus on how the providers are modeled and assembled into consumer solutions.

Note:
The Service-Oriented Architecture Modeling Language (SoaML) standard uses the term *participant* and does not distinguish between service providers and service consumers. This is because, in general, participants both provide services and use services in order to do so. It is clear from the service and request ports of a participant what is actually provided and consumed, making it unnecessary to also further characterize the participant itself.

As with all of the articles in this series, we'll use existing IBM® Rational® tools to build the solution artifacts and link them together, so that we can verify the solution against the requirements and more effectively manage change. In addition, we extend the Unified Modeling Language (UML) for services modeling by adding the Object Management Group (OMG) SoaML Profile to the UML models in IBM® Rational® Software Architect. Table 1 provides a summary of the overall process that we'll use in developing the example and the tools used to build the artifacts.

Table 1. Development process roles, tasks, and tools

Role	Task	Tools
Business executive	Convey business goals and objectives	IBM® Rational® Requirements Composer
Business analyst	Analyze business requirements	IBM® Rational® Requirements

		Composer
Software architect	Design the architecture of the solution	IBM Rational Software Architect
Web services developer	Implement the solution	IBM® Rational® Application Developer (RAD)

Service identification and specification review

Let's start by reviewing the service specifications that were identified and specified in the previous articles. Figure 1. shows the service interfaces that expose the capabilities required for processing purchase orders.

Figure 1. Capabilities for processing purchase orders

Figure 2 shows the complete `Scheduling` service interface. This service interface is a simple interface, because there is no interesting protocol for using scheduling services.

Figure 2. The Scheduling service interface

Figure 3 shows the `ShippingService` service specification.

Figure 3. Shipping service interface

This service interface is a little more complicated, because it models the interaction between a *shipper* and an *orderer* by using a simple callback-style interaction. Because this specification includes a protocol, we model it by using a service interface that defines the roles (properties) involved in the service protocol. The responsibilities of these roles are defined by their *types*, which are the interfaces provided or required by the service interface. The `shippingService` interaction owned by the `ShippingService` service specification defines the rules for how the shipper and orderer interact. This interaction will be the contract for service channels connected to a service defined by this service interface.

Figure 4 shows the `InvoicingService` service specification.

Figure 4. The InvoicingService interface

This protocol is also a bit more complicated, because the provided and required service functional capabilities must be invoked and responded to in a specific order. In this case, an activity is used to define the service protocol.

Figure 5 shows the Purchasing service specification.

Figure 5. The Purchasing service interface

The Purchasing service interface represents the functional capability specified in the original Process Purchase Order business process. It represents a service identified and designed to realize that business process. Because there is no protocol associated with this specification, we once again model this by using a simple interface.

Now we are ready to design components that provide each of these services and realize the exposed capabilities.

Service Realization

A **service** defines a set of capabilities (provided by service providers) that meet the needs of service consumers, or users. The first step in service implementation design is to provision the services. That is, we must figure out which service providers will be providing which service capabilities. This is a key part of designing an SOA, because choosing providers establishes the relationships between service consumers and providers. Therefore, this establishes both the capabilities of a system and the potential coupling between its parts.

You could put all of the operations into a single service and have a simple solution. But all clients would depend on that one service, which would result in a very high degree of coupling. Any change in the provider would result in a possible change in all consumers. This was a common problem with module libraries in the old days of **C** programming. You could also create a separate service for each functional capability, but this would result in a very complex system that would not reflect good encapsulation and cohesion. It would also be difficult to model communication between consumers and providers that follow a protocol for using a set of related functional capabilities.

In the end, deciding on the service participants is something that takes skill and can be affected by lots of compromises. Distribution can play a key role. It would be great if we could design SOA solutions independent of the participant locations, but that generally isn't very practical. Where a service is deployed in relation to consumers and other required services can have a profound effect on solution performance, availability, and security. Ignoring this in the solution architecture may result in unacceptable solution implementations down the road.

Our problem here is quite simple, so it is not hard to determine what service participant will provide or consume what services. In fact, the pools and lanes in the original Business Process Modeling Notation (BPMN) business process sketch gave a pretty good hint about what participant should provide and consume what services. The following sections provide models of service providers for all of the services shown in Figure 1. and the detailed service specifications that follow that figure. This is a fairly simple example, and many of the participants provide only one service that has only one capability. This will not generally be the case. Participants will often provide or consume (or both) many services that have many functional capabilities. This example is intentionally simple to focus on concepts without getting bogged down in the details of the example itself.

Note:
The concept of service realization in this article is a little different from that described in SOMA. In SOMA, service realization deals with architectural decisions concerning solution templates and patterns, details of SOA reference architecture, technical feasibility, and prototyping. These are beyond the scope of this series of articles,

which cover only determining which participants will provide and use what services, and how.

Invoicing

An Invoicer participant provides the Invoicing service for calculating the initial price for a purchase order. Then it refines this price when the shipping information is known. The total price of the order depends on where the products are produced and where they are shipped from. The initial price calculation can be used to verify that the customer has sufficient credit or still wants to purchase the products. It is better to verify this before fulfilling the order.

We start the design of this service provider by creating a system use case that defines its requirements and a Participant called *Invoicer* that realizes the use case, as shown in Figure 6. The Invoicer participant will be in the credit package that imports the CRM (customer relationship management) package to use the common service data (message types) definitions.

Figure 6. The initial Invoicer service provider

The Invoicer participant will provide the `InvoicingService` service. We model this by adding a `Service` to the Invoicer, which is of the type `InvoiceService` service interface. All services are typed by service interfaces that define what functional capabilities are provided and required and the protocol for using them. Figure 7 shows the Invoicer with the invoicing service added.

Figure 7. Adding an InvoicingService to the Invoicer service provider

We can see by the type of the service that it provides the Invoicing interface and requires the `InvoiceProcessing` interface. From the service's type, we know what consumers connected to the service have to do to use the service and what the Invoicer (or any other provider) has to do to implement it. Any use and implementation of a service must be consistent with the service's specification and its protocol.

The Invoicer provides the Invoicing interface, which involves two operations:

- `initiatePriceCalculation`
- `completePriceCalculation`

The Invoicer must provide a design for the implementation or method for each of these service operations that specifies how the operations will be provided. The method must also invoke the `processInvoice` operation of the `InvoiceProcessing` interface when the price calculation has been completed as specified in the service protocol. As Figure 8. shows, the Invoicer component owns two behaviors that have the same name as the provided operations.

Figure 8. Invoicer service implementations

The `completePriceCalculation` activity is the method for the `Invoicing::completePriceCalculation` service operation. It uses an opaque action to calculate the total price, and then it invokes the `InvoiceProcessing::processInvoice` operation on the invoicing port. (The target input pin of the `processInvoice` action is the invoicing service port, as shown by the activity partition containing the action.) Notice that this is consistent with the invoicing protocol as specified by the `InvoicingService` service interface.

The `initiatePriceCalculation` opaque behavior is the method for the `initiatePricesCalculation` service operation. This operation is implemented by using natural language or Java™ code captured in the body of the opaque behavior.

Production scheduling

A production scheduling participant provides the Scheduling service to determine where goods will be produced and when. This information can be used to create a shipping schedule used in processing purchase orders.

The *Productions* participant provides the Scheduling service interface through its scheduling service port, as shown in Figure 9.

Figure 9. The Productions service provider

The service operation methods are the `requestProductionsScheduling` and `sendShippingSchedule` behaviors. The details of these implementations are not shown in the diagram and may be left to be implemented by the developer by using more applicable, platform-specific languages.

Shipping

A shipping service provider provides the Shipping service interface to ship goods to a customer for a filled order. It also requires the `ScheduleProcessing` interface to request that the consumer process the completed schedule. Figure 10 shows that the Shipper service provides the shipping service as specified by the `ShippingService` service interface.

Figure 10. The Shipper service provider

In this case, we have separated the specification of the Shipper participant from its realization. This specification participant describes the architecture, both internal and external, for any realizing participant. We did this because there can be many different implementations of the Shipper participant specification, each with different additional capabilities and needs and qualities of service. We have shown one realizing participant, *ShipperImpl*. In participant service assemblies, we will use the Shipper participant specification rather than referring to ShipperImpl directly. Then,

at deployment or run time, different implementations can be substituted for this specification to achieve the desired qualities of service.

What's next

We have now finished the identification, specification, and implementation (or *realization*) design of the service participants needed to meet the business objectives. The result is a technology-neutral design model of a service solution architecture. But we still haven't created a service participant that assembles services provided by the Invoicer, Productions, and Shipper and uses them to process a purchase order. The next article in this five-part series, "Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 4. Service composition," shows how to assemble and connect these service providers and choreographs their interactions to provide a complete solution for the business requirements.

Resources

Learn

- SoaML, an OMG standard profile that extends UML 2 for modeling services, service-oriented architecture (SOA), and service-oriented solutions. The profile has been implemented in IBM Rational Software Architect.
- Daniels, John, and Cheesman, John. *UML Components: A Simple Process for Specifying Component-based Software*. Addison-Wesley Professional (2000).
- Service-oriented modeling and architecture: How to identify, specify, and realize services for your SOA by Ali Arsanjani is about the IBM Global Business Services' Service Oriented Modeling and Architecture (SOMA) method (IBM® developerWorks®, November 2004).
- IBM Business service modeling, a developerWorks article by Jim Amsden (December 2005), describes the relationship between business process modeling and service modeling to achieve the benefits of both.
- Using model-driven development and pattern-based engineering to design SOA: Part 2. Patterns-based engineering, Part 2 of a four-part IBM developerWorks tutorial series by Lee Ackerman and Bertrand Portier (2007).
- Design SOA services with Rational Software Architect, a four-part IBM developerWorks tutorial series by Lee Ackerman and Bertrand Portier (2006-2007).
- Model service-oriented architecture with Rational Software Architect: Part 3. External system modeling, Part 3 of a five-part IBM developerWorks tutorial series by Gregory Hodgkinson and Bertrand Portier (2007).
- Modeling service-oriented solutions is Simon Johnston's great article describing the approach to service modeling that drove the development of the IBM UML Profile for Software Services, the RUP for SOA plug-in (developerWorks, July 2005) and SoaML.
- SOA programming model for implementing Web services: Part 1. Introduction to the IBM SOA programming model, by Donald Ferguson and Marcia Stockton (developerWorks, June 2005), describes the IBM programming model for Service-Oriented Architecture (SOA), which enables non-programmers to create and reuse IT assets. The model includes component types, wiring, templates, application adapters, uniform data representation, and an Enterprise Service Bus (ESB). This is the first in a series of articles about the IBM SOA programming model and what is required to select, develop, deploy, and recommend programming model elements.
- Read SOA programming model for implementing Web services: Part 1. Introduction to the IBM SOA programming model, by Donald Ferguson and

Marcia Stock, to learn more about Service Data Objects, which simplify and unify the way applications access and manipulate data from heterogeneous data sources (developerWorks, June 2005).

- See Web Servoces for Business Process Execution Language for more about the BPEL 1.1 specification.
- Subscribe to the developerWorks Rational zone newsletter. Keep up with developerWorks Rational content. Every other week, you'll receive updates on the latest technical resources and best practices for the Rational Software Delivery Platform.
- Browse the technology bookstore for books on these and other technical topics.

Get products and technologies

- Download trial versions of other IBM Rational software.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Tivoli®, and WebSphere®.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Jim Amsden
Jim Amsden, a senior technical staff member with IBM, has more than 20 years of experience in designing and developing applications and tools for the software development industry. He holds a master’s degree in computer science from Boston University. His interests include enterprise architecture, contract-based development, agent programming, business-driven development, Java Enterprise Edition, UML, and service-oriented architecture. He is a co-author of =Enterprise Java Programming with IBM WebSphere (IBM Press, 2003) and of the OMG SoaML standard. His current focus is on finding ways to integrate tools to better support agile development processes. Jim is currently responsible for developing IBM Rational software’s Collaborative Architecture Management strategy and tool support.

Trademarks

Trademarked terms commonly used in developerWorks content are attributed on the

Trademarks page.