Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 2. Service specification

Skill Level: Advanced

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

14 Jan 2010

In this second article of this five-part series, we continue defining the SOA solution by modeling the specification of each service in detail. These specifications will define service interfaces between consumers and providers of the service. These service interfaces include the provided and required interfaces, the roles that those interfaces play in the service specification, and the rules or protocol for how those roles interact.

Context of this article

The first article in this series, Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 1. Service Identification, outlined an approach for identifying services that are connected to business requirements. We started by capturing the business goals and objectives necessary to realize the business mission. Next, we modeled the business operations and processes that are necessary to meet the goals and objectives. We then used the business requirements and processes to identify the required capabilities and the potential relationships between them. That provided a formal structure for identifying business-relevant capabilities that are linked to the business goals and objectives that they are intended to fulfill.

In the previous article, we also looked at how to maximize the potential of a service-oriented architecture (SOA) solution by identifying services that are business-relevant. We examined each of the capabilities by treating them as candidate services. The capabilities that passed the service litmus test were used to identify the required service interfaces. This ties the service interfaces back to the business requirements.

In this second article, we continue defining the SOA solution by modeling the specification of each service in detail. These specifications will define interfaces between consumers and producers of the service. These contracts include the provided and required interfaces, the roles that those interfaces play in the service specification, and the rules or protocol for how those roles interact.

Overview of service specifications

We are now ready to start modeling the details of the service interfaces. A service interface must specify everything that potential consumers of the service need to know to decide whether they are interested in using the service, as well as exactly how to use it. It must also specify everything that a service provider must know to successfully implement the service. At the heart of SOA is the construction of service value chains that connect user needs with compatible provider capabilities. Service interfaces define the goals, needs, and expectations of user participants, as well as the value propositions, capabilities, and commitments of provider participants. Therefore, they provide the information necessary to determine compatible needs and capabilities.

Ideally, this information is provided in a single place. This makes it easy to search asset repositories for reusable services and to get all of the necessary information without having to navigate many different documents or search for related elements. Service interfaces include at least this information:

- The name of the service, suggesting its purpose.
- The provided and required interfaces, thereby defining the functional capabilities that are provided by the service and those that it requires of its users.
  Note: This is not about how the service is implemented, but rather the interaction between the consumers and providers of this service.
- Any protocol that specifies rules for how the functional capabilities are used or in what order.
- Constraints that reflect what successful use of the service is intended to accomplish and how it will be evaluated.
- Qualities that service consumers should expect and that providers are expected to provide, such as cost, availability, performance, footprint, suitability to the task, competitive information, and so forth.
- Policies for using the service, such as security and transaction scopes for

maintaining security and integrity or for recovering from the inability to successfully perform the service or any required service.

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 (OMB) Services-Oriented Architecture Modeling Language (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

Service identification review

Let's start by reviewing the service interfaces that exposed the business capabilities needed to meet the business goals and strategies that we described in detail in "Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 1. Service identification." Figure 1. shows the service interfaces that expose the capabilities required for processing purchase orders.

Figure 1. Capabilities for processing purchase orders

The rest of this article explains how to model the details of the service interfaces. These service interfaces are an elaboration of the interfaces shown in Figure 1. They provide many of the details listed in the Overview.

When the interfaces are complete, you still will not know which service participants provide or require services described by the interfaces nor how the service capabilities are implemented, possibly by using other services. That information comes in the next article, when we cover service realization.

Types of service specifications

A service interface needs to provide this information:

- The name of the service, indicating what it is about or what it does.
- The provided and required interfaces, describing the functional capabilities of the service. Each functional capability includes:
  - Its name, which is often a verb phrase indicating what it does
  - Any required or optional service data inputs and outputs
  - Any preconditions that consumers are expected to meet before using the capability

    - Any post-conditions that **consumers** can expect and that **providers** must provide upon successful use of the capability
    - Any exceptions or fault conditions that might be raised if the capability cannot be provided for some reason, even though the preconditions have been met
- Any communication protocol or rules that determine when the capabilities can be used or in what order.
- Any capabilities that consumers are expected to provide to be able to use or interact with the service.
- Requirements that any implementer must meet when providing the service,
- Constraints that reflect what successful use of the service is intended to accomplish and how it will be evaluated.
- Qualities of service that consumers should expect and that providers are expected to provide, such as cost, availability, performance, footprint, suitability to the task, competitive information, and so forth.
- Policies for using the service, such as security and transaction scopes for maintaining integrity or recovering from the inability to successfully perform the service or any required service.

Clearly, this is a lot of information, not all of which is covered in this article. In particular, we will not be looking at qualities of services or policies. These are sufficiently complex to warrant separate articles. Furthermore, they might be specific to particular providers of a service, not the interface of a particular service itself. Instead, we'll focus on the fundamentals necessary to define and use a service.

The following subsections elaborate on each of the identified service specifications shown previously in Figure 1. The presentation order is from a very simple service interface that has no protocol, to a service interface that represents a simple request-response protocol, to a more complex service that involves a multistep protocol and interaction between the user and provider.

Scheduling service

The Scheduling service interface shown in Figure 2 is very simple. The service provides two operations: the ability to respond to a production schedule request and the ability to create a shipping schedule. These operations were created by examining the functions of the capabilities the service interface is exposing. As far as we know, in this situation there is no protocol for using these operations. Either can be used in any order.

The Scheduling service interface

The Scheduling service interface is a simple UML interface defined in the productions package. It provides two service operations. Each of these operations can have preconditions and post-conditions, and they can raise exceptions. The parameters of the service operations are required to be either service data (DataType or MessageType) or primitive types. This ensures that the parameters make no assumptions about call-by-reference or call-by-value, where the service data is located (in what address space), whether the service user or provider is operating on a copy of the data or some persistent data source, and so on. All of this is required to ensure that the service is not constrained by where it can be deployed in relation to other services. The service data is defined in the Service data model section that follows in this article.

Shipping service

The Shipping service interface is a little more complicated. A user who wants to ship products requests the shipping service. However, it could take time for the shipper to determine where the products are located, whether they are in available inventory or need to be produced, and the most cost-effective way to ship the products. Therefore, it could be a while before the shipping schedule is available. The user generally will not want to wait until the schedule is complete, because this could either prevent other work from being done in parallel or unnecessarily tie up system resources with long-running processes.

Therefore, the IT architect has decided to use a simple request response or *callback* protocol between the user and provider. The user requests the shipping and then, later on, responds to a request from the shipper to process the completed schedule. To model this protocol, we need to specify the producer and user roles, their responsibilities, and the protocol or rules for how they interact. This last part is important, because the shippers will not be able to send a schedule if they never received shipping requests.

A **service interface** tells you everything you need to know about a service. This includes the requirements that you have to meet to use the service (sometimes called the *Use* or *Usage contract* (see the Daniels and Cheesman article listed in Resources), plus the requirements that an implementer of the service has to meet (sometimes called the *Realization contract*). This is the same kind of information that you needed to capture for the business requirements, except that the subject area and level of detail are different. This is to be expected, because you are defining the specification in a service interface for how a service user and provider interact.

In this case, we use an abstract class to define the service interface and operations of the exposed capabilities, as shown in Figure 3.

Figure 3. Shipping service interface

The `ShippingService` ServiceInterface involves two roles:

- The **shipper** role is a provider role. It is responsible for fulfilling the shipping responsibilities that are given by its type, the shipping interface.
- The **orderer** role is responsible for processing the shipping schedule. This is shown by its `ScheduleProcessing` type.

It is not necessary to designate these roles as provider and user. These are arbitrary distinctions in a potentially long-running conversation, possibly involving many parties. It is also easy to see who the user and provider are by the fact that the Service specification realizes the provided shipping interface and uses the required `ScheduleProcessing` interface.

There is a connector between the shipper and orderer roles. This indicates that the protocol involves some interaction between these roles. The `shippingService` interaction that is owned by the `ShippingService` class shows what this interaction is.

The `shippingService` interaction specifies the behavioral or dynamic aspects of the interaction between the orderer and shipper roles. It shows that the orderer first sends a `requestShipping` message (or invokes the shipper's `requestShipping` operation), and then, sometime later, must respond to a `processSchedule` message from the shipper. The interaction involves two lifelines: one for the orderer and another for the shipper. These object instances are the orderer and shipper properties in the ServiceInterface. That is, the messages are exchanged between those roles through the connector between them. This is a simple, asynchronous request/response or callback pattern that is typical of many service protocols.

The `shippingService` protocol could have been specified using any UML 2 behavior: an activity, interaction, state machine, protocol state machine, or opaque behavior (code). The choice of which to use is up to the modelers, their preferred styles, or the applicability to the problem domains.

Invoicing service

The Invoicing capability shows that two operations need to be exposed to calculate invoice total. Calculating the initial and final price for an invoice involves a slightly more complex protocol between an orderer and invoicer. Obviously, the `initiatePriceCalculation` must be invoked before the `completePriceCalculation`. Then, the orderer must be prepared to process the resulting invoice.

We can capture this protocol by using a ServiceInterface that specifies the invoicer and orderer roles, their responsibilities, and the protocol (conversation or rules) for how they interact. This is just like the `ShippingService` specification, except that the interaction is more than just simple request-response. There is a sequence in which the service functional capabilities must be invoked for valid use of the service.

The `InvoicingService` service specification shown in Figure 4 captures this protocol. Notice that this service interface also implements the Invoicing use case. A use case may be used to represent the service-specific requirements. The service interface consists of two roles: invoicer and orderer. The types of these roles are the `Invoicing` realized interface and the used `InvoiceProcessing` interface, respectively. These interfaces encapsulate the responsibilities of the roles. The `InvoicingService` activity in the service specification specifies the protocol for using the service operations, the actual communication that can occur between the orderer and invoicer roles.

Figure 4. The InvoicingService interface

InvoicingService is a ServiceInterface that specifies the conversation, communication protocol, or interaction rules between an orderer and invoicer. The protocol details are captured in a UML ownedBehavior (shown using the circle-plus notation) of the class, which is used to specify the valid interaction patterns for using this service. In this case, the protocol is expressed as a UML activity.

The protocol indicates that an orderer must initiate a price calculation before attempting to get the complete price calculation. The orderer must then be prepared to respond to a request (callback, in this case) to process the final invoice. Some consumers who request the invoicing service could do more than these three actions, but the sequencing of these specific actions is constrained by the protocol. Notice that the UML ActivityPartitions in the InvoicingService activity represent the roles or properties in the InvoicingService class. An operation invocation action belonging to a partition indicates the invocation is on the role represented by the partition (the target input pin of the action is the role represented by the activity partition).

In this case, there is only one interaction between the orderer and the invoicing service, so the service specification class has only one ownedBehavor. In other situations, there could be more than one interaction between the user and provider, each using a different protocol. The service specification would have an ownedBehavior specifying the valid interaction patterns for each of these conversations.

At this point you don't know what service provider implements an InvoicingService. Nor do you know what service consumers might use it. You know only what any user has to do to use the service and what any provider must do when implementing it.

Purchasing service

Finally, there is the service interface for processing purchase orders (see Figure 5).

Figure 5. The Purchasing service interface

Like the Scheduling service interface, Purchasing is a simple interface that has only a single operation that provides the capability for processing purchase orders for a customer who is returning a completed invoice. As a side effect, the ordered items are produced (if needed) and shipped to the customer.

This service interface represents the functional capability specified in the original Process Purchase Order business process. It represents a service identified and designed from that business process.

Service data model

The Customer Relationship Management (CRM) data model defined in `package org::crm` defines all of the data used by all service operations in the service interfaces already defined. The CRM package represents the design of a Customer Relationship Management service data model that can be reused in a number of services, even services provided by different organizations. How service data is discovered and normalized and how it relates to persistent entities or physical data sources is beyond the scope of this article. What we cover here is what the service data looks like and how the model is captured.

Figure 6. The CRM services data model

Each data type shown in Figure 6 represents service data. **Service data** is data that is exchanged between service consumers and providers. The data types of parameters for service operations are typed by a DataType, PrimitiveType, or MessageType.

Note:
Service data messages are not the same as Web Services Description Language (WSDL) messages. A service operation can have any number of inputs and outputs with types of messages or primitive types. Service operations can be designed to use single input, output, and fault messages, but this is not necessary, and can result in undesirable stamp data coupling.

Service data refers to **data transfer objects (DTOs)** that can easily be exchanged between address spaces in distributed environments. Service consumers and providers make no assumptions about where the data is actually located and, therefore, they assume that they have a copy of some view on the actual persistent domain data. UML DataTypes have no identity. They are **value objects**, in that their equality is based on the value of their content, not on their identity.

Service data represent data exchanged between service consumers and providers in a possibly distributed environment. Service providers also often need access to persistent data for use in their implementation designs. The relationship between service data and persistent domain data used in service implementation designs is the responsibility of the service provider and their implementation of the service functional capabilities. Often, the service data is a selection and projection (or view) of domain data. Nonetheless, how the service data is derived from or updates domain data is up to the service implementation. **Service data objects (SDOs)** are a very useful implementation mechanism for service data messages. They also have capabilities for managing change sets and automatically committing changes to persistent stores. Service participant implementations may use these capabilities, but they do not need to be addressed in the model.

The data model uses an `<<id>>` stereotype to identify attributes that uniquely identify instances of the containing class. This will be a recurring theme throughout the services model, because Web services and the Business Process Execution Language for Web Services (BPEL), in particular, rely on business data for instance correlation or value-based object identity.

Comparison of document-centered and RPC service data

There are several SOA interaction paradigms in common use including document centric messaging, remote procedure calls (RPC), and publish-subscribe. It is beyond the scope of this article to discuss either the different characteristics of each of these paradigms or interaction styles or the circumstances in which each is most appropriate. Suffice it to say that the decision depends on cohesion and coupling, state management, distributed transactions, performance, granularity, synchronization, ease of development and maintenance, and best practices.

SoaML supports both document-centric messaging and RPC-style service data. Figure 7 shows the Purchasing and Invoicing service interfaces with the details of their operations. The Purchasing service interface uses document messaging style service data. Its operation parameters are all typed by SoaML MessageTypes `POMessage` and `InvoiceMessage`. The Invoicing service interface, by contrast, uses the data types defined in Figure 6.

The difference between the two is how the data for an operation is packaged. For operations with parameters that are typed by MessageTypes, the operation can have at most one *in* or *out* parameter. Operations that use DataType parameters can have many *in*, *out*, and *return* parameters. This allows SoaML to model the data exchanged between service consumers and providers in a manner that adheres to chosen architectural guiding principles.

Figure 7. Message-centric and RPC-style service data

Larger view of Figure 7.

What's next

In this article, we modeled the service interface of each of the identified services in detail. These interfaces indicate the provided and required interfaces, the roles that those interfaces play in the service interface, and the rules or protocol for how those roles interact in providing the service. **Service interfaces** define a contract between user requests and provider services that enable matching needs to compatible capabilities.

The next article in this five-part series, "Part 3. Service realization," explains how the services are actually implemented. The service implementation starts with deciding which participant will provide what services. That decision has important implications in service availability, distribution, security, transaction scopes, and coupling. After these decisions have been made, we can model how each service functional capability is implemented and, therefore, how the required services are actually used. Then we'll use the UML-to-SOA transformation feature included in Rational Software Architect to create a Web services solution that can be directly used in Rational Application Developer or IBM® WebSphere® Integration Developer to implement, test, and deploy the completed solution.

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.