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 contracts between consumers and producers of the service. These contracts include the provided and required interfaces, the roles those interfaces play in the service specification, and the rules or protocol for how those roles interact.

View more content in this series

Context of this article

The first article in this series, "Modeling SOA: Part 1. Service Identification" (see More in this series, upper-left), 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. We then modeled the business operations and processes that are necessary to meet the goals and objectives. Next, we viewed the business process as a service collaboration that represents a business Service Requirements contract that must be fulfilled by our eventual solution. We then used that requirements contract to help identify the required services and the potential relationships between them. That provided a formalism for identifying business relevant services that are linked to the business goals and objectives they are intended to fulfill.

In the previous article, we also looked at how to maximize the potential of an SOA solution by identifying services that are business-relevant. We designed the service topology based on the business requirements and connected the services back to the service collaborations that represented the Service Requirements contracts that the service solution must fulfill.

In this second article, we continue defining the SOA solution by modeling the specification of each service in detail. These specifications will define contracts between consumers and producers of the service. These contracts include the provided and required interfaces, the roles 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 specifications. A service specification must specify everything that a potential consumer of the service needs to know to decide if they...
are interested in using the service, as well as exactly how to use it. It must also specify everything a service provider must know to successfully implement the service. Thus, a service specification is a mediator or a contract between what consumers need and what providers provide.

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 specifications 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 consumers. Note: This is not about how the service is implemented, but rather the interaction between the consumers and providers for 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® and IBM® WebSphere® 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. Table 1 summarizes the process that we'll use to develop the example, as well as the tools that we'll use to build the artifacts. In addition, we extend the unified modeling language (UML) for services modeling by adding the IBM® Software Services Profile to the UML models in IBM® Rational® Software Architect.

Table 1: Development process roles, tasks, and tools

<table>
<thead>
<tr>
<th>Role</th>
<th>Task</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Executive</td>
<td>Convey business goals and objectives</td>
<td>IBM® Rational® RequisitePro®</td>
</tr>
<tr>
<td>Business Analyst</td>
<td>Analyze business requirements</td>
<td>IBM® WebSphere® Business Modeler</td>
</tr>
<tr>
<td>Software Architect</td>
<td>Design the architecture of the solution</td>
<td>IBM Rational Software Architect</td>
</tr>
<tr>
<td>Web Services Developer</td>
<td>Implement the solution</td>
<td>IBM® Rational® Application Developer and IBM® WebSphere® Integration Developer</td>
</tr>
</tbody>
</table>

Service identification review

Let's start by reviewing the business requirements and the identified services that are intended to meet them, as we described in detail in "Modeling SOA: Part 1. Service identification." Figure 1 shows the business requirements as a collaboration of roles in the business, role responsibilities, and the rules for how the roles interact.
This service collaboration represents a requirements contract derived from a business process that specifies what the service solution must do. It is an architecturally neutral but formal specification of the requirements that does not over-constrain the SOA solution. By architecturally neutral, we mean that the requirements contract specifies what the solution has to do but not how.

Figure 2 shows an overview of the identified service specifications that will make up the solution and use dependencies, indicating how they are intended to be used.
Lastly, Figure 3 shows how you might use those services to fulfill your business requirements.

**Figure 2. Service topology**

Lastly, Figure 3 shows how you might use those services to fulfill your business requirements.

**Figure 3. Meeting the Service Requirements contract**

This completes the identification of the services and how they relate to the business requirements. The rest of this article explains how to model the details of the service specifications. These service specifications are an elaboration of the interfaces shown in Figure 2. They provide many of the details listed in the Overview.

When the interfaces are complete, you still won't know which service participants provide or require services described by the interfaces, nor how the service capabilities are implemented, possibly using other services. That comes in the next article, when we cover service realization.
Types of service specifications

A service specification 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 their own 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 2. The presentation order is from a very simple service specification that has no protocol, to a service specification that represents a simple request/response protocol, to a more complex service that involves a multistep protocol and interaction between the consumer and provider.

Scheduling service

The Scheduling service specification shown in Figure 4 is very simple. The service provides two functional capabilities: the ability to respond to a production schedule request and the ability to create a shipping schedule. As far as we know, in this situation, there is no protocol for using these functional capabilities. Either can be used by a consumer in any order.
Figure 4. The Scheduling service specification

```
<serviceSpecification>
  Scheduling
+ requestProductionScheduling ( customerInfo : Customer, purchaseOrder : PurchaseOrder )
+ sendShippingSchedule ( schedule : Schedule )
```

The Scheduling service specification 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 (messages) 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 consumer 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.

Shipping service

The Shipping service protocol is a little more complicated. A consumer 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 consumer 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 consumer and provider. The consumer 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 consumer roles, their responsibilities, and the protocol or rules for how they interact. This last part is important, because the shipper won't be able to send a schedule if they never received a shipping request.

A service specification 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 Related topics), 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 Requirements contract for how a service consumer and provider interact.

In this case, we use an abstract class to define the service specification, as shown in Figure 5.
The ShippingService specification 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 consumer. These are arbitrary distinctions in a potentially long-running conversation, possibly involving many parties. It is also easy to see who the consumer 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 requestShipping 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 Service specification class. 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 applicability to the problem domains.

### Invoicing service

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 using an abstract class 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 6 captures this protocol. Notice that this service specification also implements the Invoicing use case. A use case may be used to represent the service-specific requirements. The service specification 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 (service or requisition functional capabilities or operations). 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 6. The InvoicingService specification

InvoicingService is a class that specifies the conversation, communication protocol, or interaction rules between an orderer and invoicer. The protocol details are captured in an `ownedBehavior` 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 requesting the invoicing service could do more than these three actions, but the sequencing of these specific actions is constrained by the protocol. Note that the `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 `ownedBehavior`. In other situations, there could be more than one interaction between the consumer 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 consumer has to do to use the service and what any provider must do when implementing it.

**Purchasing specification**

Finally, there is the service specification for processing purchase orders (Figure 7).

**Figure 7. The Purchasing service specification**

```
+ processPurchaseOrder ( customerInfo : Customer, purchaseOrder : PurchaseOrder ) : Invoice
```

Like the Scheduling service specification, 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 topology revisited**

Now that the service specifications are more fully defined, we can take another look at the service topology shown previously in Figure 3. Recall that it shows how instances of the identified services can be used to fulfill the business Service Requirements contract. But that diagram didn't show very much about the services themselves, how they are connected, or what protocols are involved in their interactions. Now that the service specifications are complete, you create a new diagram that indicates how service participants using these services might interact to fulfill the requirements. You'll be returning to this diagram in the next article in this series, "Modeling SOA: Part 3. Service realization" where it will be used as the starting point for the final integrated services solution in this example.

Figure 8 shows an abstract Order Processing component that provides a context for showing another view of the service topology. Its parts represent order processing participants that will provide or require services (or both) that are necessary to fulfill the Process Purchase Order requirements contract. We don't know exactly what these parts are yet (they don't have types), but we can specify what they need to do by indicating what roles they play in the service specifications that define how they interact and by what requirements they are intended to fulfill.
The connectors between the parts of the Order Processing component indicate anticipated interactions between the parts. The connector names correspond to the names of their contracts, which, in this case, are the protocols for the corresponding service specifications. This indicates that these parts will have to provide and require the interfaces specified by the service specification and that the parts will use those interfaces according to the service specification's protocol. How this is done is modeled in the services realization descriptions covered in the next article in this series.

We also see that the parts taken together will interact in a manner that fulfills the Process Purchase Order requirements contract. Thus, Order Processing summarizes two things:

- The service consumers and providers (or participants) need to fulfill the business requirements
- The connections and protocols that show how these participants interact to do so.

**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 Purchase Order Process model in this example (Figure 9). 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.
Each <<message>> data type in Figure 9 represents service data. **Service data** is data that is exchanged between service consumers and providers. The data types of parameters for service operations are either **messages** or **primitive types**.

**Note:**
Service data messages aren't 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.

Messages are **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. Messages have no identity. They are **value objects**, in that their equality is based on the value of their content, not on their identity.

Messages 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.

**What's next**

In this article, we modeled the service specification of each of the identified services in detail. These specifications indicate the provided and required interfaces, the roles those interfaces play in the service specification, and the rules or protocol for how those roles interact in providing the service. Service specifications define a contract between consumer requisitions and provider services that map needs to capabilities.

The next article in this five-part series, "Modeling SOA: Part 3. Service realization" explains how the services are actually implemented. The service implementation starts with deciding what component 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 WebSphere Integration Developer to implement, test, and deploy the completed solution.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSM sample model</td>
<td>RSM_sample_model.zip</td>
<td>60KB</td>
</tr>
</tbody>
</table>
Related topics

- **Modeling SOA: Part 1. Service identification** by Jim Amsden is the first in a series of five articles about developing software based on service-oriented architecture (SOA). (IBM® developerWorks®, October 2007).
- **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.
- **UML 2.0 Profile for Software Services**, also by Simon Johnston (developerWorks, April 2005), describes the UML profile for software services, a profile for UML 2.0 that allows for the modeling of services, service-oriented architecture (SOA), and service-oriented solutions. The profile has been implemented in IBM Rational Software Architect.
- **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.
- **Service Data Objects** simplify and unify the way applications access and manipulate data from heterogeneous data sources.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- 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.
- Download the trial version of IBM Rational Software Architect V7.