This fourth article of this five-part series covers how to assemble and connect the service providers modeled in "Part 3. Service realization" and choreograph their interactions to provide a complete solution to the business requirements. The resulting component will be a service participant that composes the services provided by the Invoicer, Productions, and Shipper components to provide a service capable of processing a purchase order. It also shows how this service participant fulfills the original business requirements.

View more content in this series

About this series

In previous articles in this series (see "More in this series," upper-left), we 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. Then we used the business process as a contract that helped identify the required services and the potential relationships between them. We then completed the specification of the identified services.

In the first article in this series, Part 1. Service identification, we 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 requirements contracts that the service solution must fulfill.

In the second article, Part 2. Service specification, we modeled the details of the service specification. A service specification defines everything potential consumers of the service needs to know to decided if they are interested in using the service and exactly how to use it. It also specifies everything a service provider must know to successfully implement the service.

In Part 3. Service realization we modeled the realization of the service specifications resulting in service providers: Invoicer, Productions, and Shipper. Each of these components provides services and capabilities according to the service specifications. Each provided service operation has a method that describes how the service is actually implemented. A method can be any UML
behavior, including an Activity, Interaction, StateMachine, or OpaqueBehavior. The choice is up to the modeler.

"Modeling SOA: Part 5. Service implementation," the final article in this series, uses the IBM® Rational® Software Architect UML-to-SOA transformation feature to create a Web services implementation that can be used directly in IBM® WebSphere® Integration Developer to implement, test, and deploy the completed solution.

**Context of this article**

In this article, we assemble the service providers created in the third article to use their capabilities in the method of another service provider. That is, we will be creating a new service from the composition of other services. This technique for service composition can be used recursively any number of times, across any set of concerns, or at any level of abstraction. However, there may be architectural constraints that would affect the granularity of service operations, security and performance concerns, volume of data interchange, and wire-level communication protocol and binding issues that may constrain what services are provided by what components. These issues determine the solution architecture and are not covered by these articles. See [Design an SOA solution using a reference architecture](#) for further details on this important topic.

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 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**

<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 WebSphere Integration Developer</td>
</tr>
</tbody>
</table>

**Service realization review**

Let's start by reviewing the service providers that were implemented in the previous article. Figure 1 shows the Invoicer service provider.
Figure 1. The Invoicer service provider

An Invoicer service provider provides the InvoicingProtocol service for calculating the initial price for a purchase order, and then refines this price when the shipping information is known. The total price of the order depends on where the products are produced and the location from which they are shipped. The initial price calculation may be used to verify that the customer has sufficient credit or still wants to purchase the products. It is best to verify this before fulfilling the order.

Figure 2 shows the Productions service provider.

Figure 2. Production service topology
The Productions service provider provides a 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.

Figure 3 shows the Shipper service provider.

**Figure 3. The Shipper service provider**

```
<<interface>>
Shipping
+ requestShipping ( customerInfo : Customer, shippingInfo : Manifest )
```

The Shipper service provider provides the ShippingService service to ship goods to a customer for a filled order. This service requires the ScheduleProcessing interface to request that the consumer process the completed schedule.

**Service composition**

Now that the services are all provided by some provider, we are ready to use these providers in an assembly that fulfills the original business requirements. This assembly composes and choreographs the services according to the business requirements to provide a method for the Purchasing service. We will be creating an *OrderProcessor* component that provides a purchasing service to process purchase orders. This service provider requires services defined by the InvoicingService, Scheduling, and ShippingService service specifications. We will then create an OrderProcessing component, which assembles instances of the Invoicer, Productions, and Shipper components, along with the OrderProcessor component, to implement service operation to process purchase orders.

**Order Processor service provider**

The purchase order processing service is specified by the Purchasing service specification (also shown in Figure 4) which includes the following capability (or operation):

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

This service will be provided by an OrderProcessor service provider. OrderProcessor is a component that provides a service by connecting other service providers together, which are choreographed according to a requirements contract. That is, some aspects of the provided service’s methods are designed to use other service providers in a certain way. This component provides the Purchasing interface through its purchasing service port. All customer interaction is
through this port, thereby separating customer clients from interactions that the component may have with other service consumers or providers. This limits coupling in the model, making it easier to change as the market and service consumers and providers change.

**Figure 4. The Purchasing service specification**

```
<serviceSpecification>
  Purchasing
  + processPurchaseOrder (customerInfo: Customer, purchaseOrder: PurchaseOrder): Invoice
```

The organization of the OrderProcessor component is presented in the Project Explorer view shown in Figure 5.

**Figure 5. The order management business functional area (package)**

The OrderProcessor service provider is contained in the `org::ordermanagement` package, which is used to organize services related to order management. The order management package also contains related service interfaces, service consumers, and service providers.

The OrderProcessor diagram shown in Figure 6 provides an external view of the OrderProcessor service provider and its provided and required services. (Required services are sometimes called *requisitions* to help distinguish needs from capabilities.) The external or "black box" view is what consumers of the service provider see. The component's internal structure, shown later in this example, provides an internal or "white box" view of the structure supporting the component's implementation design.
The external view is not a specification separate from an implementation; it is simply a view of some aspects of the component. If the architect or developer wanted to completely separate the specification of the service provider from its possible implementations, a specification component would be used. A specification component defines a contract between service consumers and service providers that decouples them from particular provider implementations. The specification component can be realized by many concrete components that provide the services in a manner that realizes the contract and provides acceptable qualities of service. See "Modeling SOA: Part 2. Service Specifications" for further details.

The OrderProcessor service provider component is sufficiently simple and stable that, in this example, the architect or developer decided not to use a service specification. As a result, any service consumer using the OrderProcessor component will be coupled to this particular implementation. Whether this is a sufficient design depends on how much the service will be used and how much it might change over time. Only the solution architect can decide what level of coupling is tolerable for a particular system.

The OrderProcessor component also has service ports representing requisitions for needs provided by other service providers: invoicing, scheduling, and shipping. These service providers provide services used by the OrderProcessor component to implement its provided service operations.

Each of these service interaction points is involved in a simple protocol that affects provided and required interfaces. For example, the invoicing interaction point requires the Invoicing interface to initiate price calculations and send the shipping price. However, it may take time to calculate the shipping price, so the OrderProcessor provides the InvoiceProcessing interface through its invoicing port so that the invoicing service provider can send an invoice when it is ready.
Potential coupling problems

Notice that there is a potential design problem with this example that might result in undesirable coupling. The rules for how an order processing service provider interacts with invoicing are captured in the same business process (Activity) as the rules for interfacing with production scheduling and shipping service providers. This makes it difficult to reuse invoicing, scheduling, and shipping services without duplicating the interaction protocols.

This coupling effect often results from a direct implementation of business processes as service operations. Business process models focus on the steps in an activity necessary to accomplish a specific business goal, such as efficient order processing. They are not necessarily focused on how to refactor those steps to increase cohesion, decrease coupling, facilitate reuse, minimize distributed communication overhead, handle network security, manage integrity of persistent data, and so forth. These are all IT solution concerns that must be addressed using software architectures and practiced and proven design patterns.

Using service contracts provides a way to formally capture business requirements, while at the same time allowing refactoring into architected service providers that simultaneously meet the business requirements and address IT solution concerns. The link between the architected solutions and the business requirements is through contract fulfillment, which binds the parts of the service providers with the roles that they play in the service contract. We don't address these design problems further in this example, but we do show how to link the services solution back to the business requirements that it fulfills.

Order Processor implementation design model

We have now completed the architectural structure of the service model and captured it in external views of service providers. The next thing to do is to design a method for the `processPurchaseOrder` service operation provided by the OrderProcessor component. The method must conform to any fulfilled service contracts or realized service specifications, as well as to the service specification in which the operation is defined.

Internal structure

The OrderProcessor service provider provides a single service specification, Purchasing, through its purchasing service port. This service specification specifies a single operation, `processPurchaseOrder`. A service provider must provide some method for all of its provided service operations. This example uses an Activity as the method of the `processPurchaseOrder` service operation. The details for how this is done are shown in the internal structure of the OrderProcessor component providing the service. The OrderProcessor internal structure is captured in diagrams, interfaces, classes, and activities, as shown in the Project Explorer view in Figure 7 and in the composite structure diagram in Figure 8, which follow.
The Project Explorer view shows a list of the parts of the OrderProcessor provider and a method (behavior) for each provided operation. The convention used in this example is to use a class diagram with a name that is the same as the component, and, in the package containing the component, to show its external view. This is the diagram shown in Figure 6 and at the bottom of Figure 7. A composite structure diagram of the same name as the component and contained in the component provides the internal view of the service provider's structure. This diagram is directly under the OrderProcessor service provider in Figure 7. These conventions make it easy to coordinate the external and internal views of the service participant and to scope the diagrams the same as the component.

The OrderProcessor composite structure diagram, shown in Figure 8, provides an overview of the service provider's internal structure. This again shows the parts (ports and properties) that make up the static structure of the component.
Figure 8: The internal structure of the OrderProcessor service provider

![Image of OrderProcessor internal structure]

The internal structure of the OrderProcessor component is simple. It consists of the service ports for the provided and required interfaces, plus a number of other properties that maintain the state of the service provider. The ID property is used to identify instances of the service provider. This property will be used to correlate consumer and provider interaction at runtime. The `schedule` and `shippingInfo` properties are information used in the implementation of the `processPurchaseOrder` service operation.

**Methods for provided service operations**

Each service operation provided by a service provider must be realized by either:

- Behavior (Activity, Interaction, StateMachine, or OpaqueBehavior) that is the method of the operation
- `AcceptEventAction` (for asynchronous calls) or `AcceptCallAction` (for synchronous request or reply calls) in an Activity that belongs to the component.

This allows a single Activity to (generally) have more than one concurrent entry point, and it corresponds to multiple receive activities in the Business Process Execution Language (BPEL). These `AcceptEventActions` are usually used to handle callbacks for returning information from other asynchronous `CallOperationActions`.

The OrderProcessor component has an example of both styles of service realization. The `processPurchaseOrder` operation has a method Activity with the same name. This activity, shown in Figure 9, is an owned behavior of the service provider that provides the service operation.
Figure 9: The processPurchaseOrder Service Operation Implementation

This diagram corresponds very closely to the WebSphere Business Modeler diagram for the same behavior. The InvoiceProcessing and ScheduleProcessing service operations are realized through the processInvoice and processSchedule accept event actions in the process. Notice that the corresponding operations in the interfaces are denoted as `trigger` operations to indicate the ability to respond to AcceptCallActions (similar to receptions and AcceptEventActions where the trigger is a SignalEvent). The `trigger` keyword is not part of UML 2 and is included only to highlight how these operations are realized.

Note:
These operations will not be accepted unless the processPurchaseOrder activity is running and the flow of control has reached the two accept call actions. This indicates the implementation of an operation can determine when other operations will be responded to.

**Fulfilling service contracts**

The OrderProcessor component is now complete. But there are two things left to do:

1. First, we need to tie the OrderProcessor service provider back to the business process that modeled its requirements.
2. Then we need to create a subsystem that connects service providers capable of providing the OrderProcessor’s required interfaces to the appropriate service ports.

This will result in a deployable subsystem that is capable of running. This section will deal with linking the SOA solution back to the business requirements. The next section covers the deployable subsystem.
Terminology

The IBM® Software Services Profile describes a service collaboration as “A set of services acting together in an agreed manner according to some process specification”. This is essentially a contract specifying how a set of services are connected and choreographed to achieve some objective, such as how another service should be implemented or how some business objectives would be achieved. A service collaboration could also be used to formally describe requirements that need to be met. The integration between WebSphere Business Modeler business process modeling and Rational Software Architect UML modeling is designed to exploit collaborations as a role-centered way of describing requirements. It views a WebSphere Business Modeler business process as a collaboration when it is opened in Rational Software Architect.

The terms service collaboration, service requirements contract, and business services requirements contract all mean similar things and all use collaborations to describe requirements for how a set of services are used together. The different terms simply focus on the particular use of a collaboration in a context.

Nothing in this section will change anything in how the OrderProcessor component is translated to an SOA implementation. Linking a component to services contracts describes only how the component fulfills the requirements specified by those contracts. This does not affect the service provider's implementation or how it would be translated to an SOA solution. However, the linkage is more complex than a simple dependency. It shows, specifically, what parts of the service provider play roles in the service requirements contract and how constraints on the component fulfill business constraints. This provides much richer traceability, support for fine-grained change management, and the ability to use model validation to ensure that solutions actually meet their requirements.

Figure 10 shows the requirements for the OrderProcessor service provider, using a service contract that provides a role-centered view of the business processes created by a business analyst. A collaboration use has been added to the OrderProcessor service provider to indicate the service contract it fulfills.

**Figure 10: Fulfilling the service contract**
The collaboration use, called contract in Figure 10, is an instance of the Purchase Order Process service collaboration shown in Figure 11, which follows. This specifies that the OrderProcessor service provider fulfills the Purchase Order Process business requirements. The role bindings indicate which part of the service provider plays which role in the service contract. For example, the invoicing port plays the invoicing role, and the purchasing port plays the orderProcessor role.

These role bindings are not related to the service channel connectors described in the next section. Service channel connectors are used to connect consumer requisitions to provider services in a subsystem. Role bindings specify what role the part plays in a service contract. The interpretation of a role binding can be either strict or loose. **Strict** contract fulfillment means the parts must be type-compatible with the roles that they are bound to. **Loose** contract fulfillment means that the parts are intended to play those roles according to the architect, but model validation does not and perhaps could not verify role and part compatibility. This is, perhaps, because the business service contract is incomplete or that there is only an informal sketch of the business requirements.
Figure 11: Service Requirements contract

Showing how the SOA solution fulfills the business requirements takes a little extra work to specify the contract and role bindings, but it provides a big advantage for managing change. Model queries may be used to determine which service providers fulfill what business requirements. Any change in the requirements will likely result in a change in one of the roles in the service collaboration. The modeler can then navigate directly to the parts that play those roles to determine how the service specifications that type those parts might need to change to address the change in requirements. Model validation can also be used to determine whether some role has changed, and, therefore, the parts that play that role in the SOA solution are no longer capable of performing all of the role's responsibilities. This is much more powerful than stereotyped dependencies that have no supported semantic meaning or the loose semantics of use case realizations. It is this kind of formal, verifiable connection between SOA solutions and business requirements that ensures that the solutions are business-relevant, meet the requirements, and enable agile solutions that can easily adapt to change.
Assembling the OrderProcessing subsystem

The final things to do in our SOA solution are to create an OrderProcessing subsystem that uses the service providers that we’ve implemented so far and to assemble the parts into a deployable solution.

The subsystem shown in Figure 12 represents a deployable component that connects an OrderProcessor service provider with other service providers that provide its required services. This subsystem is an assembly of components that provide all of the information necessary to deploy and run the OrderProcessor services.

Figure 12: Assembling the parts into a deployable subsystem

The OrderProcessing subsystem consists of instances of the OrderProcessor, Invoicer, Productions, and Shipper service provider components. The invoicing service of the seller component is connected to the invoicing service of the Invoicer component. This is a valid connection, because the service specification of the invoicing service of the OrderProcessor component is the conjugate of the invoicing service of the Invoicer provider. The OrderProcessor component requires the Invoicing interface, which is provided by the Invoicer service provider. It also provides the InvoiceProcessing interface for the Invoicer to receive the updated invoice.

Connecting services (instances of service specifications) means that the participants agree to interact over the connector according to the service specification. That is, they agree to follow the required protocol. The service specification defines the roles that the connected participants play in the protocol. The service channel connector between the invoicing port of the orderProcessor consumer and the invoicing port of the invoicer provider has a contract (a behavior) which is the InvoicingService behavior of the InvoicingService service specification. The name of the connector is set to the name of its contract by convention. Any interactions across this connector are required to follow that contract or protocol. These connectors formalize the use dependency in the services architecture.

Notice that the connector between the producer and seller parts has no contract. This is because there is no protocol in the Scheduling service interface; therefore, no contract is needed for the connector.
Other consumers and providers are similarly connected. The connected services can offer different binding styles. The service channel between the service interaction points can specify the actual binding style to use.

The OrderProcessing subsystem is now complete and ready to be deployed. It has specific instances of all required service providers necessary to fully implement the processPurchaseOrder service. After it is deployed, other service consumers can bind to the purchasing service of the seller OrderProcessor component and invoke the service operation.

**Summary and what's next**

We have now finished the identification, specification, and realization of the services, consumers, and providers necessary to meet the business objectives. The result is a technology-neutral but complete design model of an architected service solution.

To actually run the solution, we need to create a platform implementation that is consistent with the architectural design decisions captured in the services model. We could create that solution by hand, using the model as a guide. But this would be tedious, error-prone, might take a while, and would require a highly skilled developer to ensure that the architectural decisions were implemented correctly. It is certainly possible to create the solution by hand, and having the model as a guide would be very helpful. But having a complete, formal, verified model gives us an opportunity to exploit model-driven development (MDD) to create a solution skeleton from the model and then complete the detailed coding in a platform-specific programming environment. That is the subject of the next and last article in this series, "Modeling SOA: Part 5. Service Implementation." In that article we use the Rational Software Architect UML-to-SOA transformation feature to create a Web services solution that can be used directly in 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>

© Copyright IBM Corporation 2007

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)