IBM

developerWorks®

Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 4. Service composition

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

28 January 2010

This fourth article of this five-part series covers how to assemble and connect the service participants modeled in "Part 3. Service realization" and then choreograph their interactions to provide a complete solution to the business requirements. The resulting service participant will assemble the services provided by the Invoicer, Productions, and Shipper participants in a services value chain 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 "View more content in this series"), 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 to identify the required capabilities and services. We then completed the specification of the identified services, allocated them to participants, and implemented the participants.

In the first article in this series, "Part 1. Service identification," we looked at how to maximize the potential of a service-oriented architecture (SOA) solution by identifying services that are business-relevant. We identified needed capabilities based on the business requirements and exposed these capabilities 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 needs to know to decided whether they are interested in using the service and exactly how to use it. It also specifies everything that a service provider must know to successfully implement the service.

In "Part 3. Service realization," we modeled the realization of the service interfaces, which resulted in Invoicer, Productions, and Shipper participants. Each of these participants provides services and realizes capabilities according to the service interface. 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.

In this article, Part 4, we look at how service participants can be assembled by connecting compatible requests and services through service channels. These assemblies are how service participants are composed, together, to provide solutions, including implementations of other services.

In "Part 5. Service implementation," the final article in this series, we will use the IBM® Rational® Software Architect UML-to-SOA transformation feature to create a Web service 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 participants created previously, in the third article, to use their capabilities in the methods of another participant. That is, we will be creating a new service from the assembly or 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 might be architectural constraints that 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 the IBM® developerWorks® article titled "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 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 service modeling by adding the Object Management Group Service-Oriented Architecture Modeling Language (OMG SoaML) Profile to the UML models in IBM Rational Software Architect. Table 1 provides a summary of the overall process that we use in developing the example and the tools used to build the artifacts.

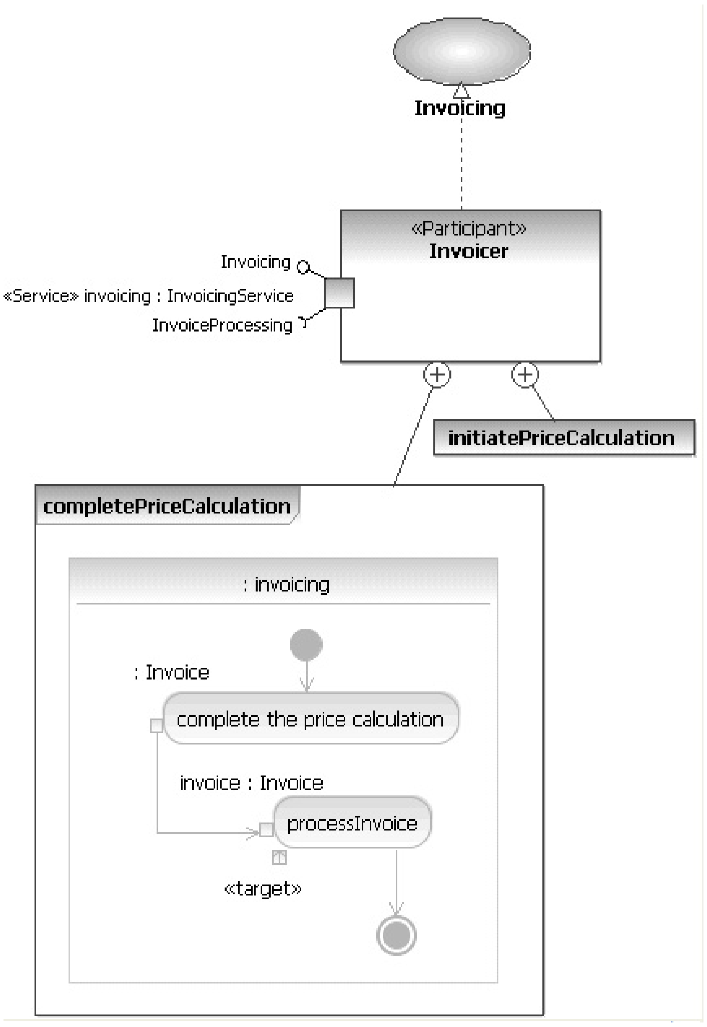
Table 1. Development process roles, tasks, and tools

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 realization review

Let's start by reviewing the service participants that were implemented in the previous article. Error: Reference source not found shows the Invoicer service provider.

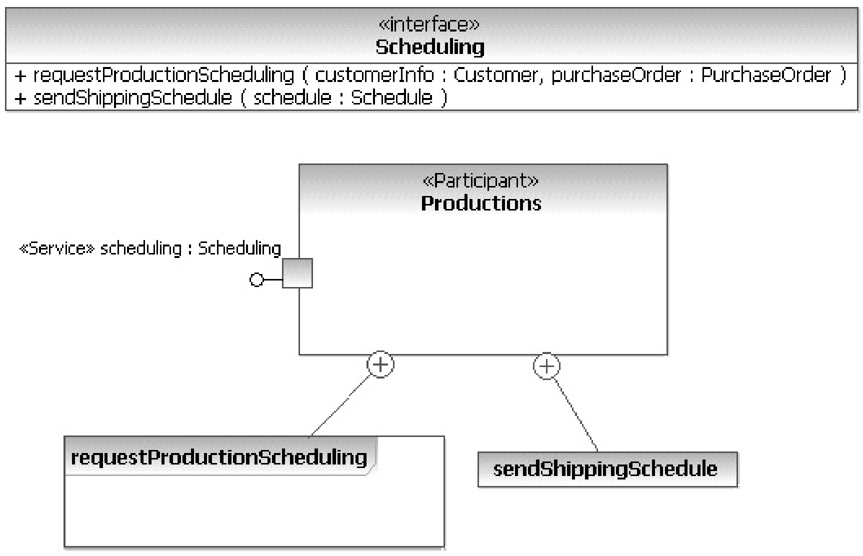
Figure 1. Invoicer service implementation



An Invoicer service participant provides the invoicing 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.

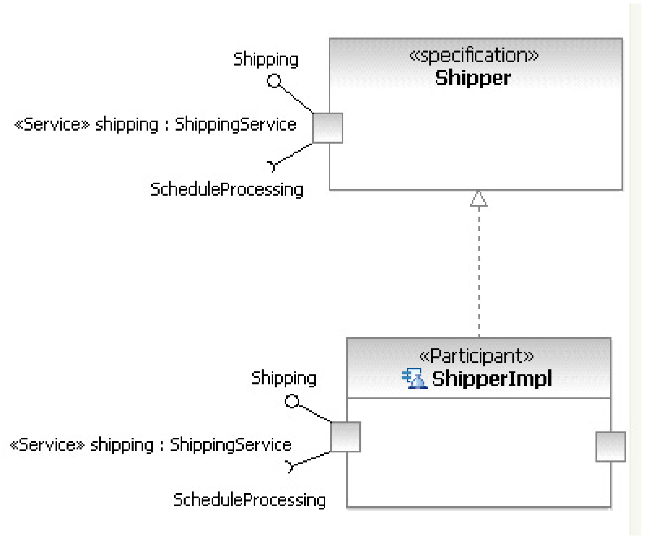
Figure 2. The Productions service provider


«interface»
Scheduling
+ requestProductionScheduling (customerInfo : Customer, purchaseOrder : PurchaseOrder)
+ sendShippingSchedule (schedule : Schedule)
«Participant»
Productions
«Service» scheduling : Scheduling
requestProductionScheduling
sendShippingSchedule

The Productions service participant 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



The Shipper service provider provides the shipping 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 participant that provides a purchasing service to process purchase orders. This service provider requires services defined by the InvoicingService, Scheduling, and ShippingService service interfaces. We will then create a Manufacturer participant, which assembles instances of the Invoicer, Productions, and Shipper participants, along with the OrderProcessor component, to implement the service operation to process purchase orders.

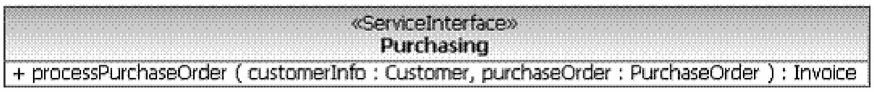
Order Processor participant

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

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

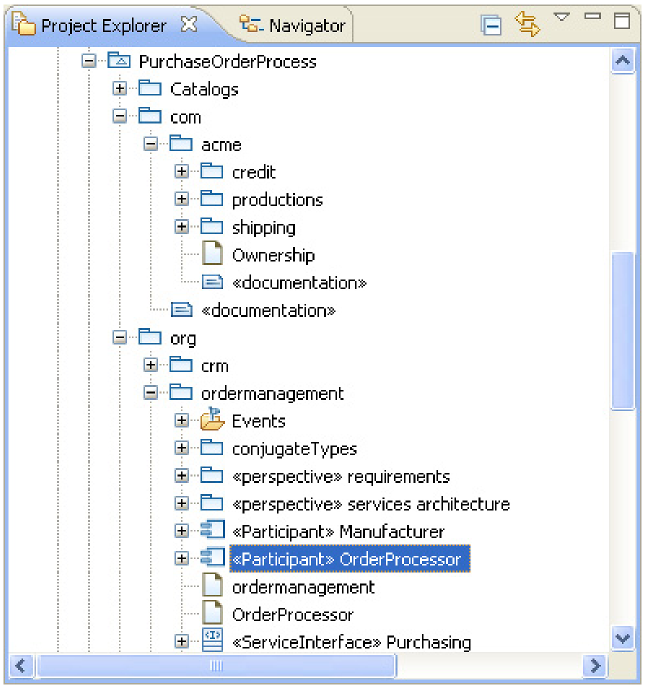
This service will be provided by an *OrderProcessor* participant. OrderProcessor is a participant that provides a service by connecting other service providers together, which are choreographed according to a requirements contract. That is, some aspects of the provided service's methods are designed 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 might have with other service consumers or providers. This limits coupling in the model, thus making it easier to change as the market and service consumers and providers change.

Figure 4. The Purchasing service interface


«ServiceInterface»
Purchasing
+ processPurchaseOrder (customerInfo : Customer, purchaseOrder : PurchaseOrder) : Invoice

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

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

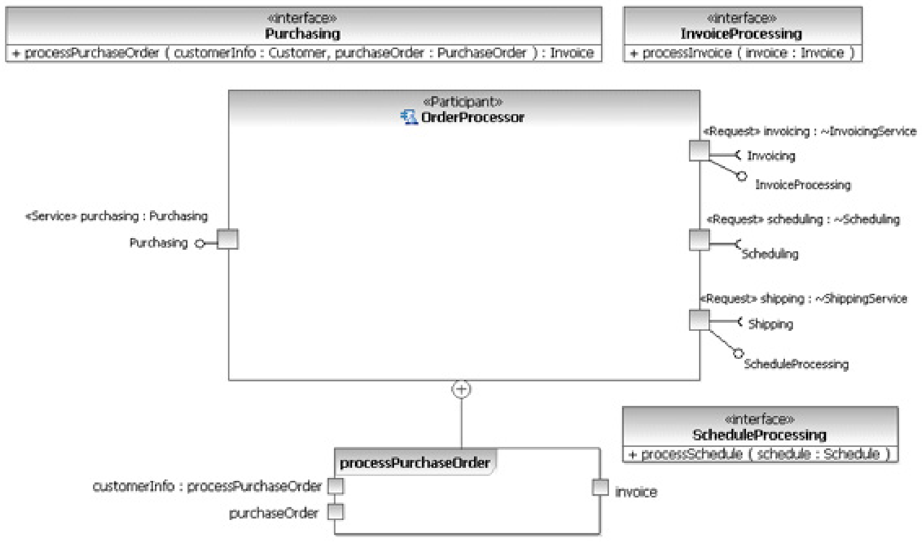


```
PurchaseOrderProcess
  Catalogs
  com
    acme
      credit
      productions
      shipping
      Ownership
      «documentation»
    «documentation»
  org
    crm
    ordermanagement
      Events
      conjugateTypes
      «perspective» requirements
      «perspective» services architecture
      «Participant» Manufacturer
      «Participant» OrderProcessor
      ordermanagement
      OrderProcessor
      «ServiceInterface» Purchasing
```

The OrderProcessor participant 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 and other participants.

The OrderProcessor diagram shown in Figure 6 provides an external view of the OrderProcessor service provider and its service and request ports. Required services are denoted as Requests to distinguish participant needs from capabilities. The external or *black box* view is what other participants see. The participant's internal structure, shown later in this example, provides an internal or *white box* view of the structure that supports the participant's implementation design.

Figure 6. The OrderProcessor



«interface»
Purchasing
+ processPurchaseOrder (customerInfo : Customer, purchaseOrder : PurchaseOrder) : Invoice
«interface»
InvoiceProcessing
+ processInvoice (invoice : Invoice)
«Participant»
OrderProcessor
«Request» invoicing : ~InvoicingService
Invoicing
InvoiceProcessing
«Request» scheduling : ~Scheduling
Scheduling
«Request» shipping : ~ShippingService
Shipping
ScheduleProcessing
«Service» purchasing : Purchasing
Purchasing
«interface»
ScheduleProcessing
+ processSchedule (schedule : Schedule)
processPurchaseOrder
customerInfo : processPurchaseOrder
purchaseOrder
invoice

Larger view of Figure 6.

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, as was done for the Shipper participant in Figure 3. A specification component defines a contract between service consumers and service providers that decouples them from particular provider implementations. The specification participant 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 Part 2. Service specification for further details.

The OrderProcessor participant is sufficiently simple and stable. Therefore, 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 business or solution architect can decide what level of coupling is tolerable for a particular system.

The OrderProcessor component also has request ports for specifying needs that will be provided by other participants: invoicing, scheduling, and shipping.

Each of these service and request interaction points can involve a protocol that specifies how the provided and required interface operations are used. For example, the invoicing request port requires the Invoicing interface to initiate price calculations and send the shipping price. However, it can take time to calculate the shipping price, so the OrderProcessor provides the InvoiceProcessing interface through its request port so that the invoicing service provider can send an invoice when it is ready.

Order Processor implementation design model

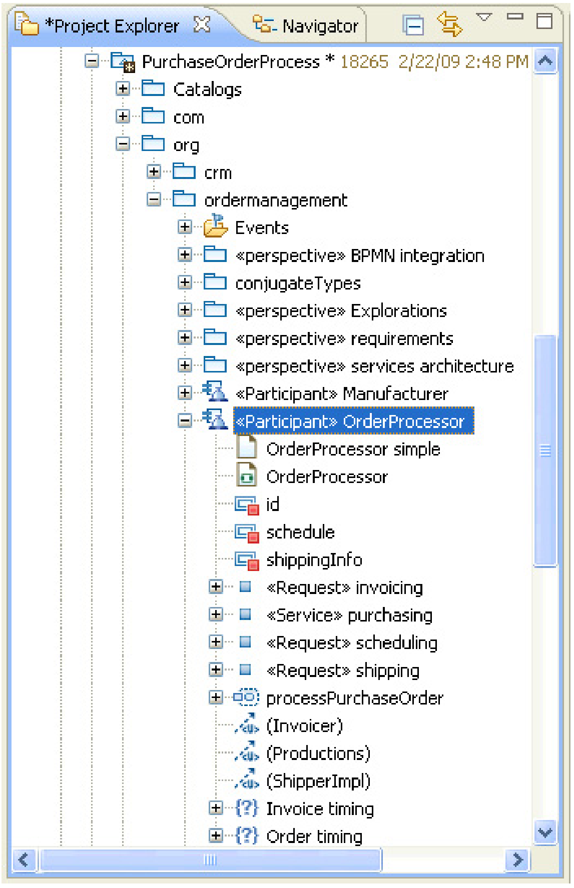
We have now completed the architectural structure of the service model and captured it in external views of service providers. The next thing to do is to design a method for the processPurchaseOrder service operation provided by the OrderProcessor component. The method must conform to any fulfilled service 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 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.

The Project Explorer view in Figure 7 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 participant to show the external view that was shown in Figure 6.

Figure 7. The organization of the OrderProcessor

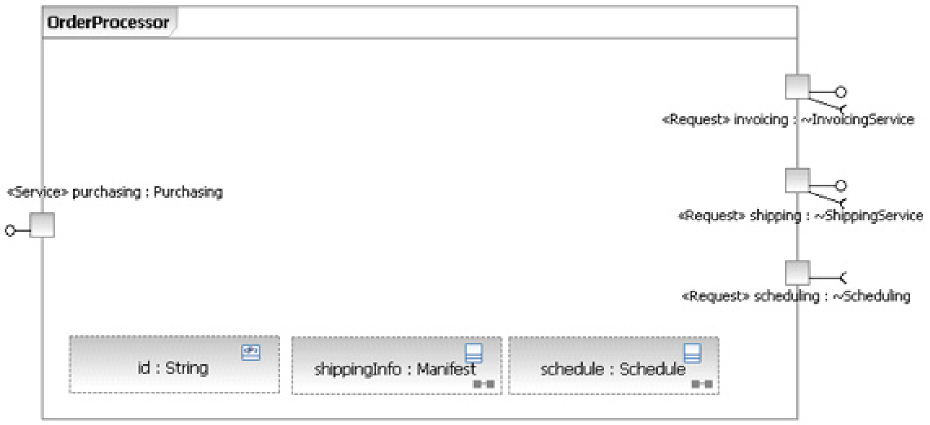


- PurchaseOrderProcess * 18265 2/22/09 2:48 PM
  - Catalogs
  - com
  - org
    - crm
    - ordermanagement
      - Events
      - «perspective» BPMN integration
      - conjugateTypes
      - «perspective» Explorations
      - «perspective» requirements
      - «perspective» services architecture
      - «Participant» Manufacturer
      - «Participant» OrderProcessor
        - OrderProcessor simple
        - OrderProcessor
        - id
        - schedule
        - shippingInfo
        - «Request» invoicing
        - «Service» purchasing
        - «Request» scheduling
        - «Request» shipping
        - processPurchaseOrder
        - (Invoicer)
        - (Productions)
        - (ShipperImpl)
        - Invoice timing
        - Order timing

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 8. 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 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 participant.

Figure 8. The internal structure of the OrderProcessor service provider


OrderProcessor
«Request» invoicing : ~InvoicingService
«Service» purchasing : Purchasing
«Request» shipping : ~ShippingService
«Request» scheduling : ~Scheduling
id : String
shippingInfo : Manifest
schedule : Schedule

Larger view of Figure 8.

The internal structure of the OrderProcessor component is simple. It consists of the service and request ports for the provided and required interfaces, plus several 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 run time. The `schedule` and `shippingInfo` properties are information used in the implementation of the processPurchaseOrder service operation.

Note:
In this example, all of the services are stateless, but the participants that provide and use services often are not.

Methods for provided service operations

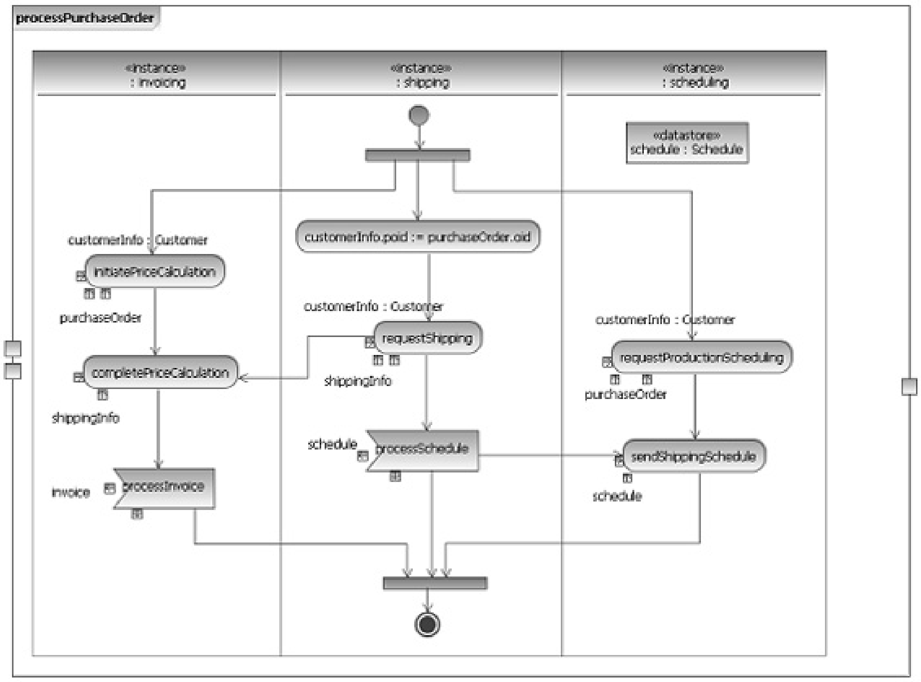
Each service operation provided by a service provider must be realized by either a behavior or an action:

- A Behavior (Activity, Interaction, StateMachine, or OpaqueBehavior) that is the method of the operation
- An 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 IBM® Rational® Requirements Composer Business Process Modeling Notation (BPMN) business process 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 receptions to indicate the ability to respond to call events sent by other participants.

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 that the implementation of an operation can determine when other operations will get responses.

Assembling the participants

The OrderProcessor component is now complete. But to be executable (implemented), in any environment, manual or automated, the OrderProcessor participant must be connected to participants that are capable of fulfilling its requests. Otherwise, operations invoked on the request ports would go nowhere, because there are no participants to perform them. There are two things left to do:

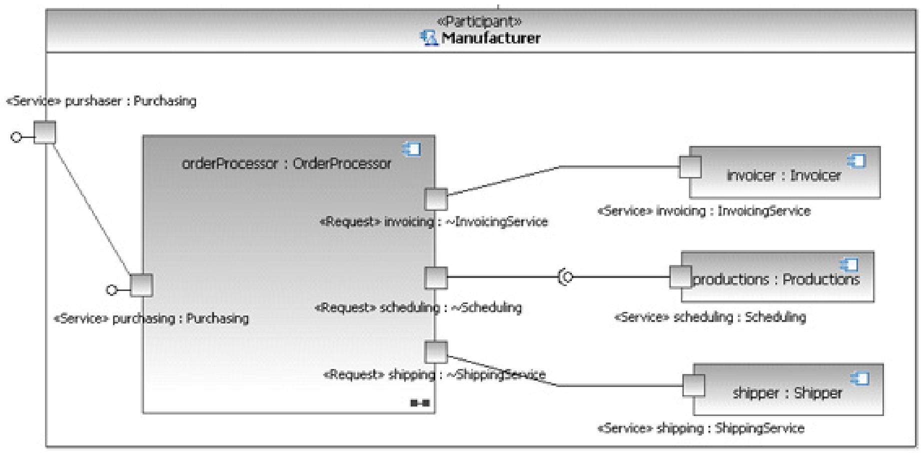
1. First, we need to create a participant that assembles the other participants that are capable of providing the OrderProcessor's requests to the appropriate services.
2. Then we need to tie the OrderProcessor service provider back to the service architecture that modeled its requirements.

This will result in a complete assembly of potentially active participants that could run in some execution environment (including manual processes).

This section will deal with assembling the participants in a new *Manufacturer* participant. The next section covers tying the solution back to the service architecture that it adheres to.

The Manufacturer participant shown in Figure 10 represents a complete participant that connects an OrderProcessor service provider with other service providers that provide its requested services. This participant is an assembly of other participants that provide all of the information necessary to complete order processing and purchasing services.

Figure 10. Assembling the Manufacturer parts


«Participant»
Manufacturer
«Service» purshaser : Purchasing
orderProcessor : OrderProcessor
«Request» invoicing : ~InvoicingService
«Request» scheduling : ~Scheduling
«Request» shipping : ~ShippingService
«Service» purchasing : Purchasing
invoicer : Invoicer
«Service» invoicing : InvoicingService
productions : Productions
«Service» scheduling : Scheduling
shipper : Shipper
«Service» shipping : ShippingService

Larger view of Figure 10.

The Manufacturer participant is an assembly of parts that are references to instances of the OrderProcessor, Invoicer, Productions, and Shipper participants. The invoicing service of the seller participant is connected to the invoicing service of the Invoicer participant. This is a valid connection, because the service interface of the invoicing request of the OrderProcessor participant is the same as the invoicing service of the Invoicer participant.

Connecting services and requests means that the participants agree to interact over a service channel according to the service interfaces. That is, they agree to follow the required protocol. The service interface defines the roles that the connected participants play in the protocol. The service channel connector between the invoicing request port of the orderProcessor consumer and the invoicing service port of the Invoicer provider establishes the contract between the participants. Any interactions across this service channel are required to follow the service interface protocols.

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.

Service value chain

The service channel that connects the orderProcessor request to the Invoicer service establishes a connection in a value chain. The request of the orderProcessor participant represents the goals that it is intended to achieve through its invoicing request port, the reasons that it needs to achieve those goals, and the qualities of service expectations that deemed acceptable. All of this information is captured in the service interface used to define the request. Conversely the service of the Invoicer represents the value propositions that it offers through its invoicing service port, the capabilities that it has in order to support the achievement of that value, and the qualities of service the participant is willing and able to commit to fulfilling. This exchange of value is composed in the Manufacturer participant where it recursively contributes to its value propositions, capabilities, and commitments available through its purchasing service port.

Key point

The essence of SOA is the establishment of these service value chains in a manner that meets business objectives while minimizing the coupling between participants. Reducing coupling between participants results from developing a good service architecture, and it is what leads to business agility.

The Manufacturing participant is now complete and ready to be implemented and deployed in some execution environment. It has references to specific instances of all required service participants necessary to fully implement the processPurchaseOrder service operation. After it is deployed, other service consumers can bind to the purchasing service of the Manufacturer participant and invoke the service operation.

Adhering to the service architecture

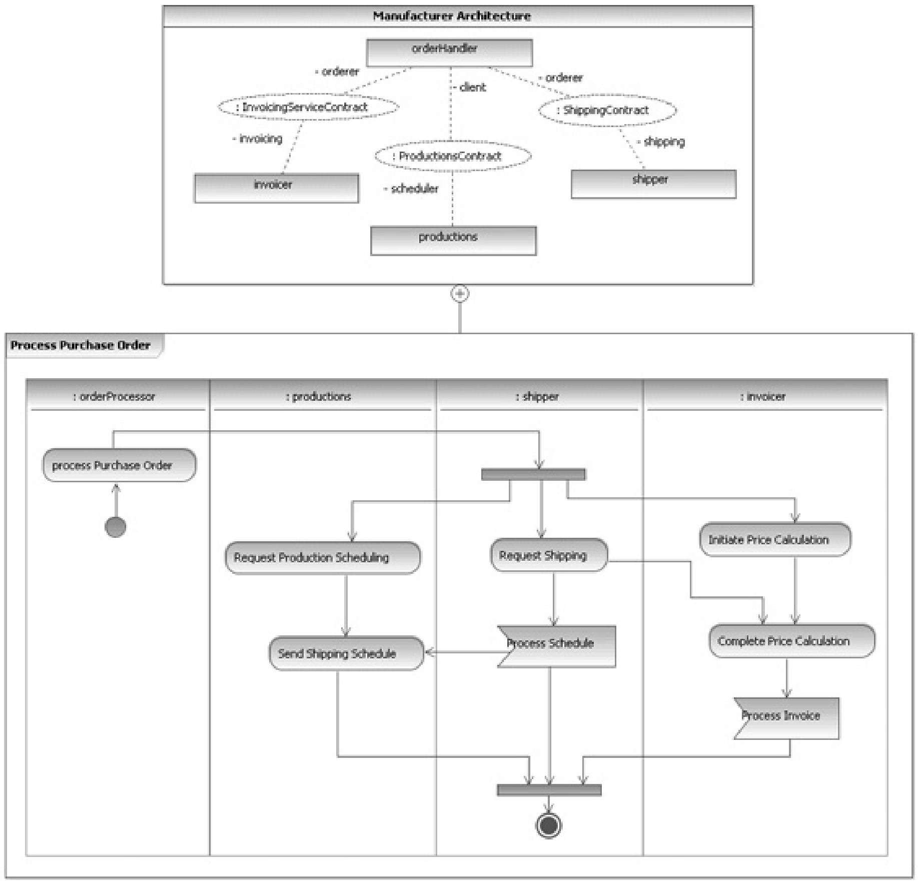
The last thing to do in this example is to tie our solution back to the service architecture that we designed in the first article, Part 1. Service identification. Our solution is already tied to the business motivation and strategy, because the services interfaces identified in Part 1 expose the business capabilities that realize the business strategies and goals. In a broader example, we could use the OMG Business Motivation Model (BMM) to capture a more complete picture of the business motivation and strategy. Such a model could include not only the business goals, but the influencers that motivate the business to change that identify those goals, the strategies and tactics that support the achievement of those goals, and the assessments of potential effect on the business in terms of risk reduction or increased reward. This could be used to tie services in our solution directly to their effects on the business, thereby identifying the value that justifies their construction.

Nothing in this section will change anything in how the Manufacturer participant is translated to any particular platform implementation. Linking a participant to service architecture describes only how the participant fulfills the requirements specified by that architecture. This does not affect the service provider's implementation or how it would be translated to a platform 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 architecture and how constraints on the participant fulfill architectural and 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 11, which is from Part 1. Service identification, shows the service architecture for processing purchase orders. A collaboration use has been added to the Manufacturer participant to indicate the service architecture that it fulfills.

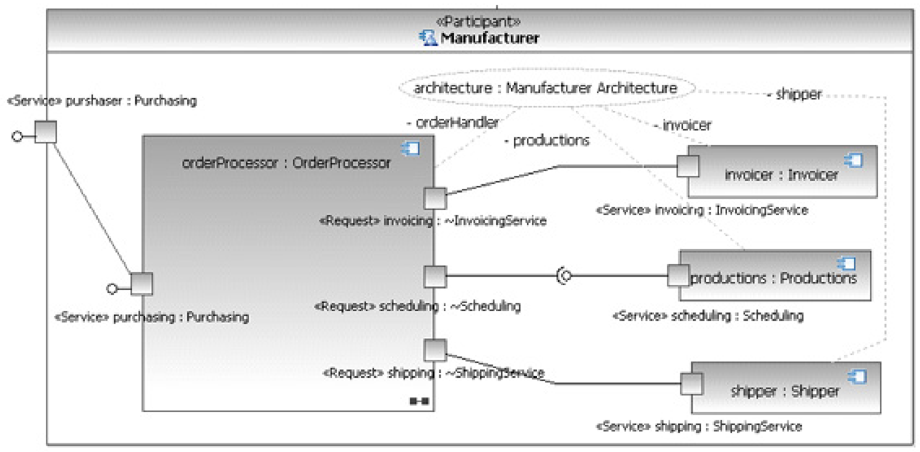
Figure 11. Services architecture for processing purchase orders



Larger view of Figure 11.

The collaboration use, called `architecture` in Figure 12, is an instance of the Manufacturer architecture service architecture from Part 1. This specifies that the OrderProcessor service provider adheres to that service architecture. The role bindings indicate the roles played by participants in the service architecture. For example, the invoicing participant plays the Invoicer role, and the orderProcessor participant plays the orderHandler role.

Figure 12. Adhering to the service architecture


«Participant»
Manufacturer
«Service» purshaser : Purchasing
architecture : Manufacturer Architecture
- shipper
- orderHandler
- productions
- invoicer
orderProcessor : OrderProcessor
invoicer : Invoicer
«Request» invoicing : ~InvoicingService
«Service» invoicing : InvoicingService
productions : Productions
«Service» purchasing : Purchasing
«Request» scheduling : ~Scheduling
«Service» scheduling : Scheduling
«Request» shipping : ~ShippingService
shipper : Shipper
«Service» shipping : ShippingService

Larger view of Figure 12.

These role bindings are not related to the service channel connectors described in the previous section. *Service channel connectors* are used to connect consumer requests to provider services in a participant assembly. *Role bindings* specify what role the part plays in a service architecture. The interpretation of a role binding can be either strict or loose.

- *Strict* means that the parts must be type-compatible with the roles that they are bound to.
- *Loose* 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 service architecture is incomplete or only an informal sketch.

Showing how the SOA solution fulfills the business architecture takes a little extra work to specify the service architecture and role bindings. But it provides a big advantage for managing change. Model queries can be used to determine which participants fulfill what business requirements. Any change in the requirements will likely result in a change in one of the roles in the service architecture. The modeler can then navigate directly to the parts that play those roles to determine how the service interfaces that specify types for 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 whether the parts that play that role in the SOA solution are no longer capable of performing all of the role's responsibilities as a result. 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 ensure that the solutions are business-relevant, meet the requirements, and enable agile solutions that can easily adapt to change.

Summary and what's next

We have now finished the identification, specification, and realization of the services and the participants that provide and use them to meet the business objectives. The result is a platform- and technology-neutral but complete design model of service solution architecture.

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 with soaML, the Service-Oriented Architecture Modeling Language: Part 5. Service implementation" (see "More in this series"). In that article, we use the Rational Software Architect UML-to-SOA transformation feature to create a Web service solution that can be used directly in 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

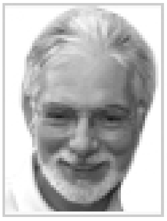
- Download trial versions of other IBM Rational software.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Tivoli®, and WebSphere®.

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.