developerWorks®

Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 5. Service implementation

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

04 February 2010

The example in this final article of this series uses 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.

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 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 interfaces resulting in participants Invoicer, Productions, and Shipper. 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 Part 4. Service Composition we looked at how services 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 this article, the final article of this series, we'll use 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

The steps in the previous articles created a complete SOA solution model that meets the business requirements. Therefore, we know what requirements that this solution architecture fulfills and what might need to change when the requirements change.

To deploy and run this solution, we need to create an actual implementation that is consistent with the architectural and design decisions captured in the model. We could create that solution by hand, using the model as a guide. But this would be tedious, error-prone, and time-consuming, and it 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 one or more solution skeletons from the model and then complete the detailed coding in a platform-specific programming environment. That is the subject of this article. We'll use the Rational Software Architect UML-to-SOA transformation feature to create a Web service that can be used directly in WebSphere Integration Developer to implement, test, and deploy the completed solution.

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

Table 1. Development process roles, tasks, and tools

Role	Task	Tools
Business executive	Convey business goals and objectives	IBM® Rational® Requirements Composer
Business analyst	Analyze business requirements	IBM Rational Requirements Composer
Software architect	Design the architecture	IBM Rational Software Architect

Web services developer	Implement the solution	IBM® Rational® Application Developer
Web services integration	Deploy a Web service	IBM WebSphere Integration Developer

But let's start by reviewing the services and service participants that were created in the previous article, Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 1. Service identification (see "More in this series").

Service specification and realization review

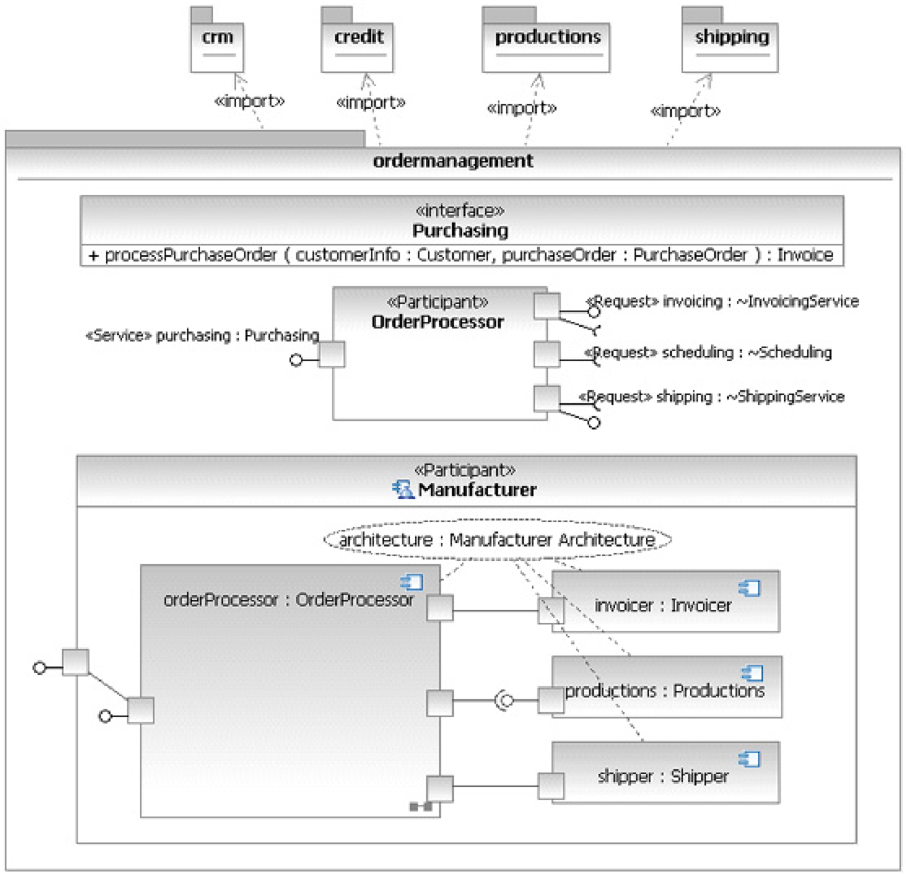
Figure 1 provides an overview of the completed example. The ordermanagement package contains the key model elements: the Purchasing ServiceInterface and the OrderProcessor and Manufacturer participants. The ordermanagement package imports model elements from other packages. The customer relationship management (CRM) package includes the domain data model that defines the persistent entities that are used to implement the services, as well as the service data MessageTypes that are used to exchange information between the service participants. The service data is essentially a view (a selection and projection) on the domain data to meet the needs of the specific services.

The credit, productions, and shipping packages define the Invoicer, Productions, and Shipper participants, respectively, as well as their provided service interfaces. OrderProcessor is another participant that provides a purchasing service and uses services for invoicing, scheduling, and shipping.

The Manufacturer participant is an assembly of references to instances of other participants that are connected in a service value chain. That is, the orderProcessor has its requests fulfilled by the services provided by the invoicer, productions, and shipper parts. Service interactions, events, and data exchanges occur across the service channels that connect the participating parts. The Manufacture participant also provides a purchasing service, but it does so by delegating to its orderProcessor internal part. All of the service operations have methods that indicate how the services are actually carried out and how the consumed services are actually used.

The Manufacturer participant also adheres to the Process Purchase Order service architecture defined in the first article, "Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 1. Service Identification." The parts of the Manufacture participant are bound to the participant roles that they play in the service architecture to show how the architectural pattern is followed and instantiated. This ensures that the solution is following enterprise architecture guiding principles, and it provides traceability between the parts of the architecture and the parts of the solution.

Figure 1. Overview of processing purchase orders


crm
credit
productions
shipping
«import»
«import»
«import»
«import»
ordermanagement
«interface»
Purchasing
+ processPurchaseOrder (customerInfo : Customer, purchaseOrder : PurchaseOrder) : Invoice
«Participant»
OrderProcessor
«Service» purchasing : Purchasing
«Request» invoicing : ~InvoicingService
«Request» scheduling : ~Scheduling
«Request» shipping : ~ShippingService
«Participant»
Manufacturer
architecture : Manufacturer Architecture
orderProcessor : OrderProcessor
invoicer : Invoicer
productions : Productions
shipper : Shipper

Service modeling best practices

UML 2 modeling helps improve our understanding of underlying systems by enabling us to abstract away extraneous details. Nonetheless, modeling isn't a panacea, and meaningful model diagrams still can require expertise to create and understand. This is a natural consequence of the need to support a very general model of computing that can be used for a wide range of application domains and levels of abstraction, and it also represents the semantics of several platform-specific execution models. In addition, the types of actions or styles of activity models might need to be constrained to allow efficient transformation of those models to the target execution platform.

In this case, the target platform is Web services and the IBM SOA programming model supported by WebSphere Integration Developer. This includes business objects (XSD), interfaces (WSDL), module assemblies (SCDL, an IBM predecessor of Open SCA, sometimes called "Classic SCA"), processes (BPEL4WS: Business Process Execution Language for Web Services), and Java™ components. To support transformations of UML models to this particular Web services platform, we need to follow service modeling best practices. We start by extending UML with the Object Management Group (OMG) SoaML standard profile to model an SOA solution architecture, and then use a particular modeling style to produce models that can be translated to Web services.

UML is typically customized by using profiles. A **profile** defines a number of stereotype classes that can extend one or more UML metaclasses with additional properties and relationships. Profiles are applied to UML models to make these extensions available. These applied profiles generally support two purposes in model-driven architectures:

- The first, and most general, use of profiles is to customize the abstractions that are intended to be supported. In this case, we have applied the OMG SoaML profile to our Purchase Order Process model to extend UML to support services modeling. Many of the stereotypes in that profile simply clarify how the UML metaclasses are used to model an SOA solution and to restrict the kinds of things that can be done in UML in order to ensure that the SOA models reflect SOA principles. For example, the `<<Participant>>` stereotype is used to indicate a UML component that is modeling a service consumer or provider. For an example of a restriction, SoaML requires all interfaces realized or used by a `<<Participant>>` component to be handled through service and request ports (UML ports), not directly. This is to reduce the coupling between the connected consumers and providers by having dependencies on the ports, not the component as a whole. Then, when some interface changes, only that port has to be examined to respond to the change, not everything connected to the component.
- The second purpose of profiles in model-driven architecture (MDA) is to support platform-specific markings. These markings consist of additional stereotypes and properties that "mark up" a model with information necessary to translate the model to a particular platform. For example, a UML package may need to have a specific Uniform Resource Identifier (URI) when translated to a Web service container.

Sometimes, these two purposes of profiles are combined. For example, the extensions to UML to support relational data modeling consist of a single profile that both extends UML for entity-relational-attribute (ERA) data modeling and provides the markings necessary to translate the UML domain models into IBM® Info Sphere® Data Architect logical data models (Lames).

In other cases, one profile can be used for modeling support while another is used to drive translation. For example, consider the case of modeling SOA designs that are to be implemented in Java™ Platform, Enterprise Edition (JEE). The latter application can be assisted by applying both the SoaML profile and the Enterprise JavaBeans™ (EJB) transformation profile to the same model. The SoaML profile stereotypes would be applied to model elements to support services modeling; whereas, the EJB transformation profile stereotypes would be applied to model elements to guide the execution of the Rational Software Architect UML-to-EJB transformation as it generates implementation code. Of course, the same SOA model could also be translated to Web services by using the Rational Software Architect UML-to-SOA transformation. The UML-to-SOA transformation would generate Web services by keying off of the SoaML profile markup. It would ignore the markings for the EJB transformation.

The next sections describe some of the modeling best practices for SOA models that are intended to be translated to Web services, in particular Web services as implemented in the IBM® SOA Programming Model and as supported by WebSphere Integration Developer.

Data modeling

The type of a service operation parameter should be either a UML Primitive Type, DataType, or an SoaML `<<MessageType>>` DataType. Modelers should make no assumptions about the location of the data, call-by-value, or call-by-reference semantics nor any implicit concurrency management facilities. Assume that service implementations are working on a transient copy of the data that might have been transferred, transformed, or both from its original source. This ensures minimal data coupling between service consumers and service providers.

SoaML supports either Remote Procedure Call (RPC) style or document-centered style parameters for service operations. RPC style supports multiple input and output parameters. Document-centered style allows, at most, one input and one output. Use SoaML MessageType parameters to indicate document-centered style.

Services modeling

As specified in the SoaML profile, a service participant must realize or use all interfaces through service ports, never directly. This ensures proper decoupling between service participants connected to the component.

Activity modeling

A concrete service provider models the methods for its service operations by providing a behavior for each operation.

Note:
A **concrete** service provider is a component that is not abstract nor a `<<specification>>` component.

Any behavior can be used, but if the model target platform is Web services, it is convenient to use an activity that can easily be translated to BPEL4WS. For example, the activity model of the Order Processor service provider component is the method for the processPurchaseOrder operation. There are several things about this activity that require further explanation:

- The signature for a method behavior must match its specification operation.
- Input and output pins on the actions are ordered starting at the lower-right corner of the containing action, and they proceed clockwise around the action to the lower-right side. This pin ordering corresponds to the order of the parameters of the called operation, with the target input pin being the first pin and the type of the operation (if any) being the last output pin corresponding to the return result. The target input pin represents the target object to which the action request is sent (for example, the classifier that owns the operation).
- The input and output pin types are not generally set, because these can be derived from the corresponding parameter.
- This activity does not use object flows to simplify the creation of the diagram. Instead, the names of the input and output pins on the actions are labeled by a parameter, variable, or structural feature in the context classifier (the class owning the activity) that is in scope. UML 2 supports either, and which one to use is a matter of preference.

- The activity parameter nodes (on the left and right edges of the activity) are not used. Instead, the parameters of the activity (which must correspond to the parameters of its specification operation) are referenced directly on input and output pins where needed. These activity parameter nodes would be used if object flows were used.
- The activity partitions are set to represent the service ports or parts of the component containing the activity. All invocations are made and all events are accepted through these parts. The partitions are not named in this case, because the represented property provides sufficient information to identify the partition.
- The target input pins of the call operation actions need not be set. As an alternative, the activity partition represents the service port where the calls in that partition are invoked. These are called <<instance>> partitions in UML 2 and have well-defined semantics. The target input pins could have been set too, but this would be redundant.
- The `returnInformation` pin of the Accept Call actions is handled the same as the target input pin of a call operation action. It is also the port represented by the activity partition, and it represents the interaction point through which the call will be accepted.
- Assignment expressions are shown with opaque actions, where the name of the action contains an assignment expression that references variables, parameters, and structural features in scope. The `lvalue` and `rvalue` in the assignment statement is separated by a colon plus an equal sign (`:=`).
- Guard expressions on object and control flows are Java or XPath Boolean expressions that reference variables, parameters, and structural features in the activity scope.
  - The *data* on an object flow is referenced by the name of the flow.
  - The *type* of the data on an object flow is determined by its source.

These conventions are used to simplify activity modeling, to simplify the activity diagrams, and to correspond better with the BPEL that will be generated from them.

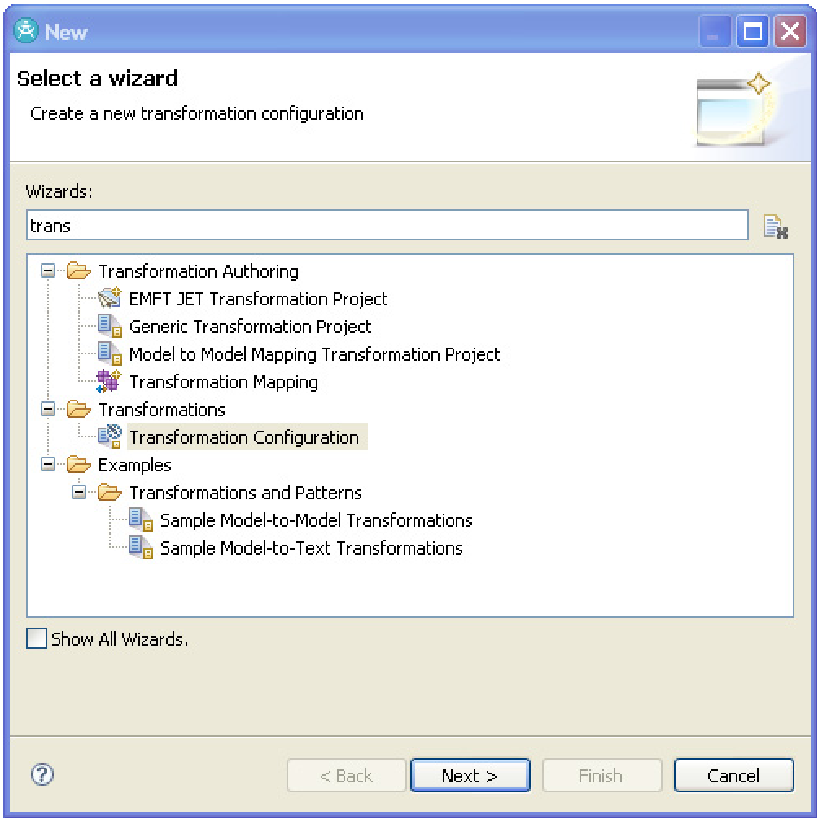
Translating to Web services

Transformations require using a transformation configuration.

Configuring the transformation

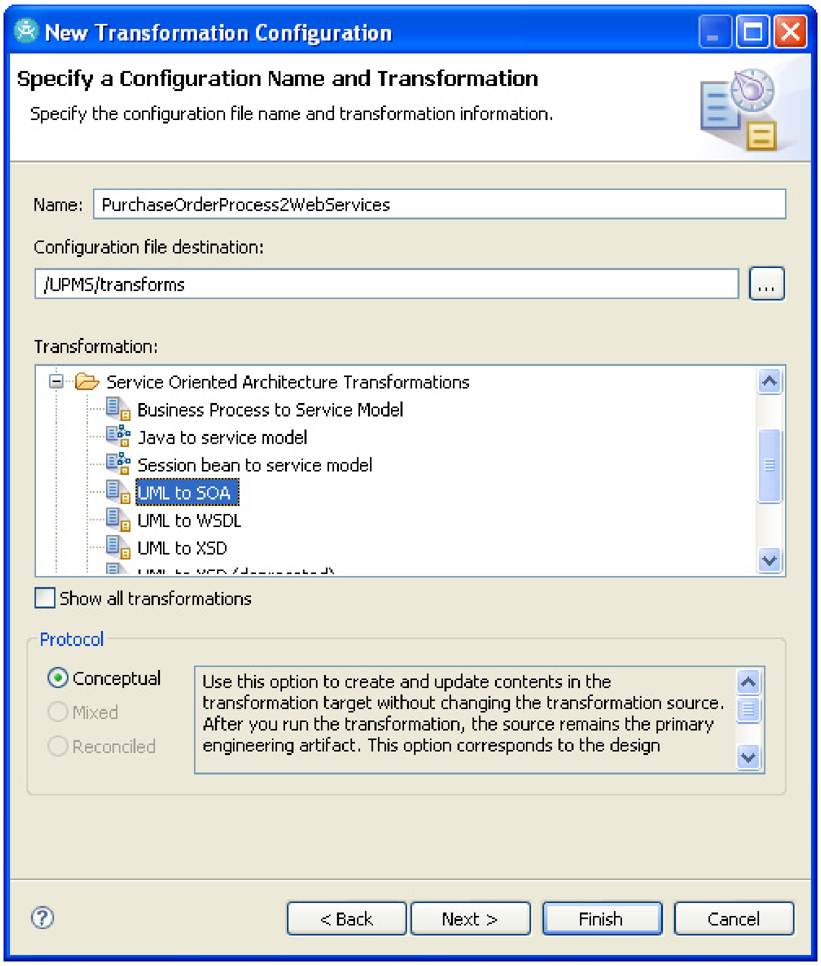
You can create a transformation by selecting **File > New > Other > Transform Configuration** (see Figure 2).

Figure 2. Creating a new transformation configuration


New
Select a wizard
Create a new transformation configuration
Wizards:
trans
Transformation Authoring
EMFT JET Transformation Project
Generic Transformation Project
Model to Model Mapping Transformation Project
Transformation Mapping
Transformations
Transformation Configuration
Examples
Transformations and Patterns
Sample Model-to-Model Transformations
Sample Model-to-Text Transformations
Show All Wizards.
< Back
Next >
Finish
Cancel

We'll be using a UML-to-SOA transformation for this example, as Figure 3 shows.

Figure 3. Selecting the UML-to-SOA transformation



Specify a Configuration Name and Transformation

Specify the configuration file name and transformation information.

Name: PurchaseOrderProcess2WebServices

Configuration file destination: /UPMS/transforms

Transformation:

- Service Oriented Architecture Transformations
  - Business Process to Service Model
  - Java to service model
  - Session bean to service model
  - UML to SOA
  - UML to WSDL
  - UML to XSD

[ ] Show all transformations

Protocol: [✓ Conceptual] Mixed Reconciled

Use this option to create and update contents in the transformation target without changing the transformation source. After you run the transformation, the source remains the primary engineering artifact. This option corresponds to the design

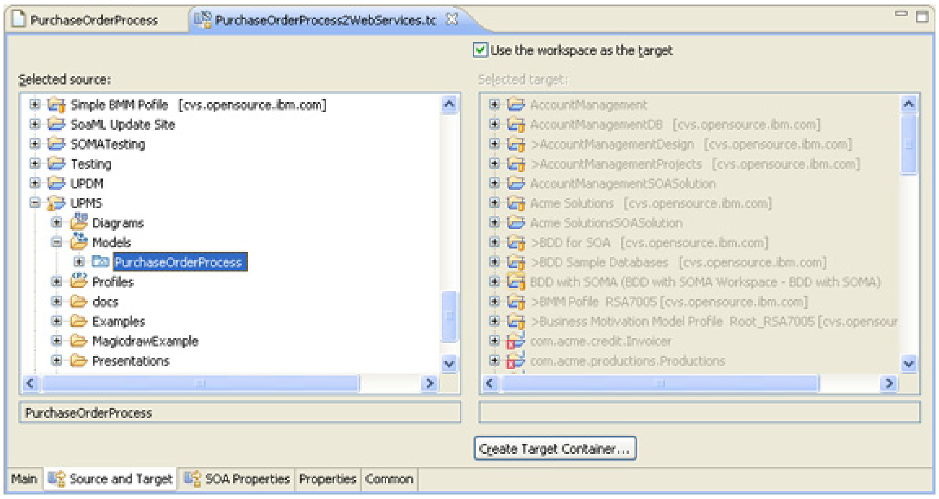
The configuration of most transformations consists of three basic parts:

1. Selecting the transformation source elements
2. Selecting (or creating and then selecting) the target elements
3. Configuring the transformation properties

The permissible source elements are defined by the particular transformation selected. Generally, it is best to transform only whole models, not individual parts of models. This ensures that the models, which represent individual versioned resources, are treated as *compilation units* with respect to model transformation. This simplifies model management and transformation dependency handling by ensuring that changes in resources in the workspace correspond to builder and transformation deltas that should be processed to ensure that derived artifacts are synchronized with their source elements. For example, you wouldn't think of selecting an individual method in a Java class and compiling just that method, and then inserting it into the byte codes for the rest of the class. This would be impossible to manage in a rapidly changing environment, and it would require you, rather than the builders and compilers, to know all of the dependencies that might need to be compiled as a result of the change.

In this example, the PurchaseOrderProcess model is the source, and the workspace is chosen as the target. All of the translated model elements are placed in new WebSphere Integration Developer projects (Figure 4).

Figure 4. Configuring the transformation sources and targets



PurchaseOrderProcess | PurchaseOrderProcess2WebServices.tc

[✓] Use the workspace as the target

Selected source:

```
+ Simple BMM Pofile  [cvs.opensource.ibm.com]
+ SoaML Update Site
+ SOMATesting
+ Testing
+ UPDM
- UPMS
    + Diagrams
    - Models
        + PurchaseOrderProcess
    + Profiles
    + docs
    + Examples
    + MagicdrawExample
    + Presentations
```

PurchaseOrderProcess

Selected target:

```
+ AccountManagement
+ AccountManagementDB  [cvs.opensource.ibm.com]
+ >AccountManagementDesign  [cvs.opensource.ibm.com]
+ >AccountManagementProjects  [cvs.opensource.ibm.com]
+ AccountManagementSOASolution
+ Acme Solutions  [cvs.opensource.ibm.com]
+ Acme SolutionsSOASolution
+ >BDD for SOA  [cvs.opensource.ibm.com]
+ >BDD Sample Databases  [cvs.opensource.ibm.com]
+ BDD with SOMA (BDD with SOMA Workspace - BDD with SOMA)
+ >BMM Pofile  RSA7005 [cvs.opensource.ibm.com]
+ >Business Motivation Model Profile  Root_RSA7005 [cvs.opensour
+ com.acme.credit.Invoicer
+ com.acme.productions.Productions
```

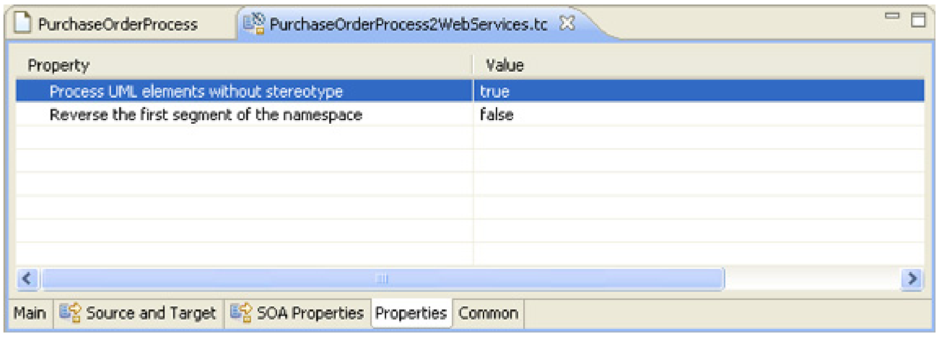
Create Target Container...

Main | Source and Target | SOA Properties | Properties | Common

Larger view of Figure 4.

The transformation configuration parameters represent transformation options that are not included in markings in the model. Typically, these control overall options rather than options that apply to a particular model element. The UML-to-SOA transformation has only a few transformation options, as shown in Figure 5.

Figure 5. Configuring the transformation properties



PurchaseOrderProcess | PurchaseOrderProcess2WebServices.tc

Property	Value
Process UML elements without stereotype	true
Reverse the first segment of the namespace	false

Main | Source and Target | SOA Properties | Properties | Common

Process UML elements without stereotypes set to a value of *True* means that the SoaML profile is actually optional. Data types, components, and activities are translated to the Web services solution without requiring any stereotypes, or the stereotypes may be left off particular model elements that don't need their additional properties. However, it is a good modeling practice to apply the SoaML stereotypes to simplify the services model.

This completes the configuration of the transformation of the PurchaseOrderProcess model to a Web services solution. Projects will be placed in the workspace.

Running the transformation

Executing the `PurchaseOrderProcess2WebServices` transformation is quite simple:

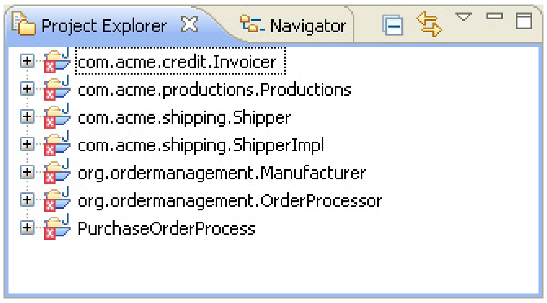
1. Select the **PurchaseOrderProcess2WebServices.tc** transformation configuration file that was created in the previous section.
2. From the pop-up menu, select **Transform > UML to SOA**.

The selected transformation in the transformation configuration is executed, thereby transforming the source model into the Web services artifacts and placing the resulting WebSphere Integration Developer projects into the workspace. You can then start WebSphere Integration Developer and import these existing project into your WebSphere Integration Developer workspace to complete, deploy, and test the results.

Tip:
If the model changes, you will need to rerun this transformation.

The results of the transformation are shown in Figure 6, using the Modeling perspective in Project Explorer.

Figure 6. Transformation results



com.acme.credit.Invoicer
com.acme.productions.Productions
com.acme.shipping.Shipper
com.acme.shipping.ShipperImpl
org.ordermanagement.Manufacturer
org.ordermanagement.OrderProcessor
PurchaseOrderProcess

Examining the result

The UML-to-SOA transformation configuration places the resulting elements in a number of Eclipse projects. These projects are either WebSphere Integration Developer library or module projects, as described in the following subsections.

- The *library* projects contain the business objects, interfaces, and module exports that are shared by other projects.
- The *module* projects contain a module implementation for each participant in the UML services model.

You can import these projects into your WebSphere Integration Developer workspace.

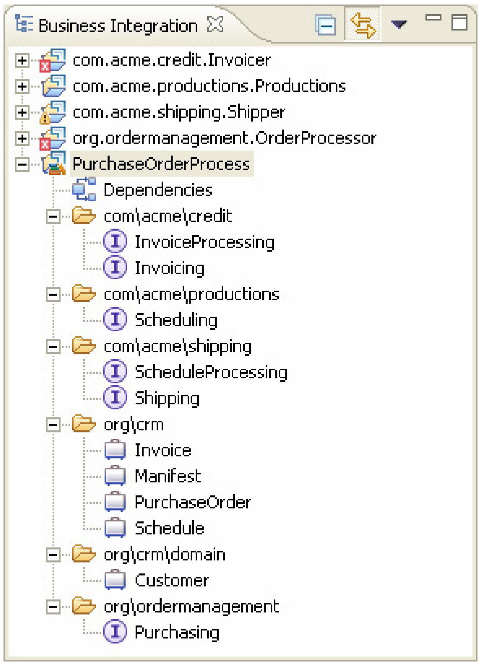
Tip:
You might find it helpful to turn off Automatic Build until after all of the projects have been imported, just to speed up the imports.

Model and libraries

Each model in the transformation configuration's selected source is translated to a WebSphere Integration Developer library with the same name as the model. This library contains an XSD element for each class and data type in the source models and a WSDL definition for each UML interface. These libraries define the business objects and interfaces used by all the WebSphere modules generated from the components in the transformation configuration's selected source.

Figure 7 shows the imported generated library and module projects in WebSphere Integration Developer with the PurchaseOrderProcess expanded to show the generated business objects and interfaces. Notice that the folders and name space in WebSphere correspond directly to the package structure in the UML services model. This ensures consistent name space management and reuse support across the resources and tools.

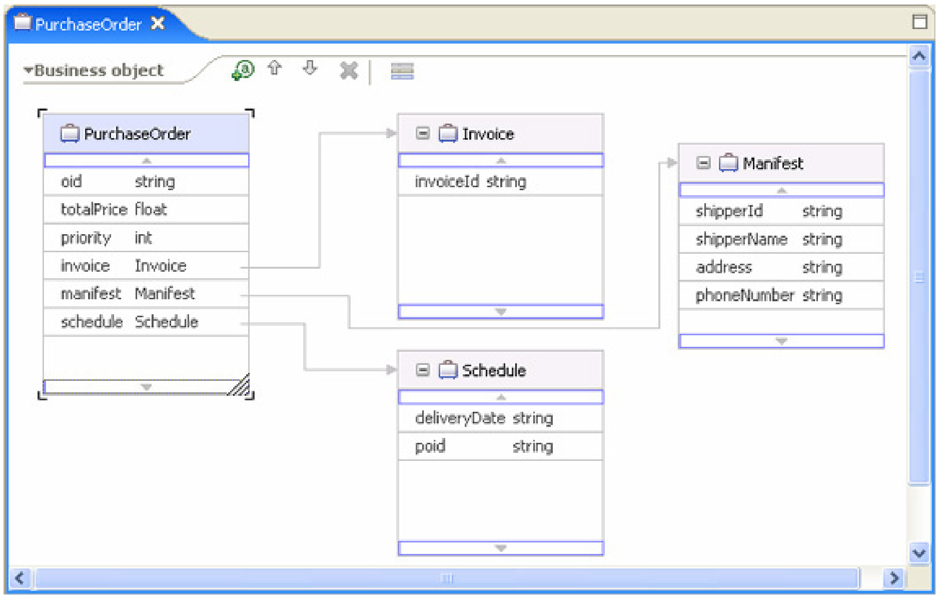
Figure 7. The ProcessPurchaseOrder library and its business objects and interfaces



```
Business Integration
com.acme.credit.Invoicer
com.acme.productions.Productions
com.acme.shipping.Shipper
org.ordermanagement.OrderProcessor
PurchaseOrderProcess
    Dependencies
    com\acme\credit
        InvoiceProcessing
        Invoicing
    com\acme\productions
        Scheduling
    com\acme\shipping
        ScheduleProcessing
        Shipping
    org\crm
        Invoice
        Manifest
        PurchaseOrder
        Schedule
    org\crm\domain
        Customer
    org\ordermanagement
        Purchasing
```

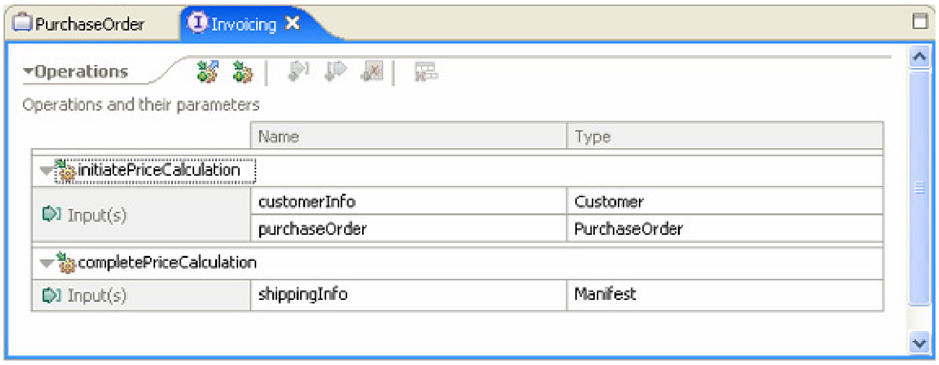
Let's take a closer look at the business objects and interfaces and compare them with their UML source elements. Figure 8 uses the WebSphere Integration Developer Business Object editor opened on the PurchaseOrder business object to show the XSDs generated from the service data model shown in Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 3. Service Realization. As you can see, the XSDs correspond very closely to their source data types. Click the figure to see the generated source.

Figure 8. The XSDs generated from the service data model



Each UML interface is transformed into a WSDL portType. The WSDL generated for the Invoicing interface in Figure 8 is shown in Figure 9. Click the figure to see the generated WSDL source. Again, the WSDL looks very similar to the UML interface.

Figure 9. WSDL generated for the Invoicing interface



Operations and their parameters

	Name	Type
initiatePriceCalculation		
Input(s)	customerInfo	Customer
	purchaseOrder	PurchaseOrder
completePriceCalculation		
Input(s)	shippingInfo	Manifest

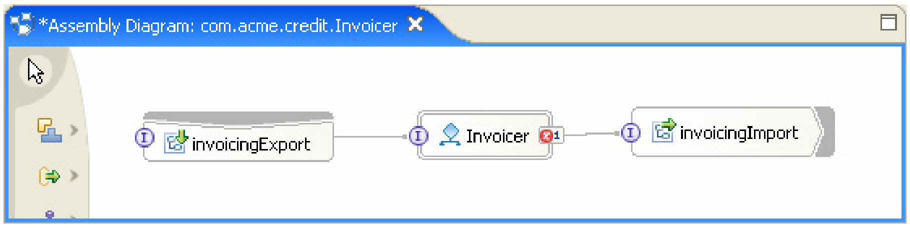
Components and module assemblies

Each service provider component in the UML services model is transformed into a WebSphere Integration Developer module. There is no Web services standard yet for assembling service consumers (sometimes called *users*) and providers. Therefore, WebSphere Integration Developer uses a proprietary, early version of Service Component Architecture (SCA). Module assemblies are captured in **.component** files that use Service Component Description Language (SCDL), which is an XML document language for service component assemblies. Several companies are collaborating to develop a standard for SCA. See the Open SOA Web site for further information.

The UML-to-SOA transformation creates WebSphere Integration Developer modules for each participant in order to maximize reuse. SCA modules cannot be assembled from other modules. However, they can import services from other modules and use them indirectly. Therefore, connections between participants in UML are implemented as bindings between module imports and exports in WebSphere Integration Developer. For example, consider the Invoicer service provider shown in Figure 1.

Figure 10 shows the corresponding WebSphere Integration Developer module assembly. Module imports and exports are created for each service and request port. SCA cannot support both provided and required interfaces for the same interaction point, thus separate import and export elements are created for service and request ports that provide and require interfaces. The invoicingExport service exports the provided Invoicing interface; whereas, invoicingImport imports the required InvoiceProcessing interface of the invoicing service port of the Invoicer service provider.

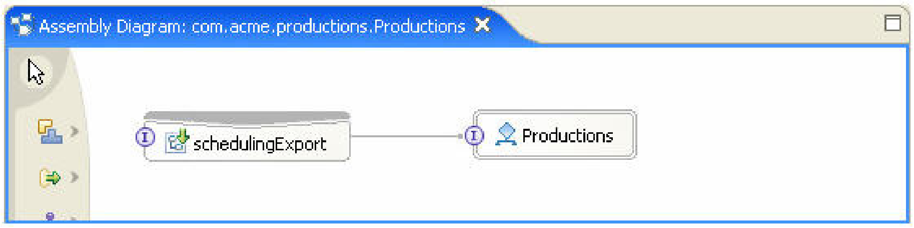
Figure 10. Invoicer module assembly



Notice the module name. A module is an eclipse project, but because a module is a reusable element, it must manage name collisions just like any other reusable element. The convention used by the UML-to-SOA transformation is to create module project names based on the fully qualified name of the service provider component, as determined by its containing package. This results in long module names that may cause some runtime problems on Windows platforms due to restrictions in the length of URLs. These module names can be easily refactored into shorter names as long as name conflicts are managed properly in the context in which they are reused.

The Productions component results in another module assembly, which is shown in Figure 11. This module has no import, because the service port doesn't require any interfaces.

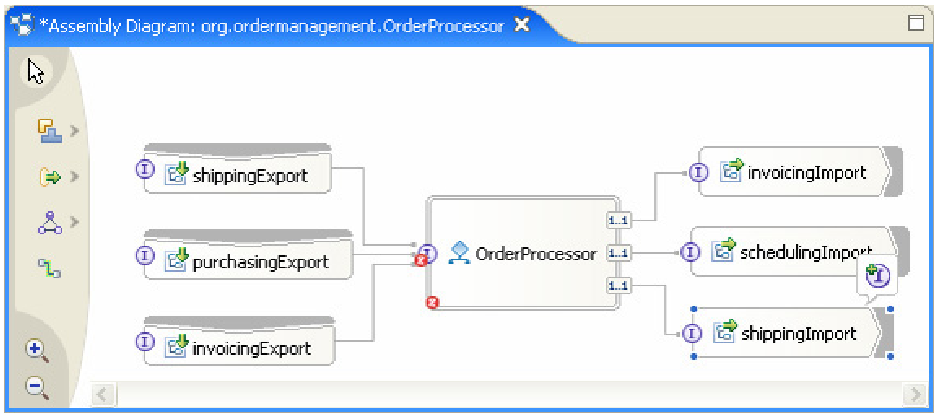
Figure 11. Productions module assembly



Both of these modules use a BPEL process to actually implement the services provided by their corresponding service provider components. The details for how this is done are shown in the next section.

Take a look at Figure 12 to see the module assembly created for the OrderProcessor component.

Figure 12. OrderProcessor module assembly



The OrderProcessor service provider provides the purchasing service and has requests for invoicing, scheduling, and shipping services. The service channels that connect the consumer and provider components in Figure 1 are implemented as imports in the OrderProcessor module bound to the exports of the corresponding service providers. This allows effective module reuse in WebSphere Integration Developer and keeps the UML services models independent of the evolution of SCA. When SCA is standardized, the UML services models won't have to change; only the transformation will need to be updated.

Activities and BPEL processes

Each functional capability (operation) provided by a service provider must be implemented somehow. The implementations are either designed in UML by using a method behavior for each operation or Accept Call actions in some other behavior. The latter method provides additional decoupling between consumers and providers by allowing the provider to determine when it is willing and able to respond to a request, rather than having to respond immediately when the operation is invoked by the service consumer.

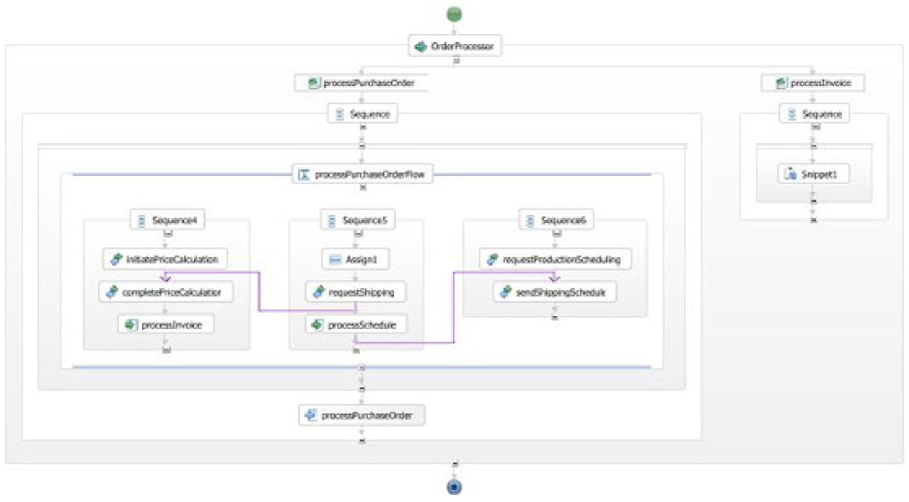
WebSphere Integration Developer SCA takes a different approach. Each export must be wired to a component in the assembly that provides an implementation for the operations in that interface. Components in SCA each have implementation types:

- Human Task
- Java
- Process
- Rules Group
- State Machine

The OrderProcessor service provider in Figure 12 provides a single functional capability through its purchasing service to process a purchase order. The implementation of this operation was an activity in the UML service model. The UML-to-SOA transformation creates a BPEL process from that activity and uses it as the implementation of the exported operation.

The BPEL process in figure 13 is very similar to the processPurchaseOrder activity in the OrderProcessor participant. The details for how the UML activity is transformed into a BPEL process can be found in the Help section of Rational Software Architect.

Figure 13. OrderProcessor process component implementation



Larger view of Figure 13.

Decoupling interface and implementation

As already described, WebSphere Integration Developer SCA implements provided capabilities by wiring an export, which provides the interfaces to a component that implements the provided operations. Because a component has an implementation type, all of the operations provided by all interfaces of that export have to be implemented the same way. This couples the implementation type to all of the interfaces of an export. (An export can be wired to and implemented by only one component.) If the developer wants to change the implementation type of a particular operation (for instance, from a human task to an automated service implemented in Java), the interfaces must be refactored to allow different exports to be wired to different component implementation types. This, in turn, would require changes to all consumers of those interfaces. This coupling of interface design to implementation type could inhibit the very business agility that SOA solutions are intended to support.

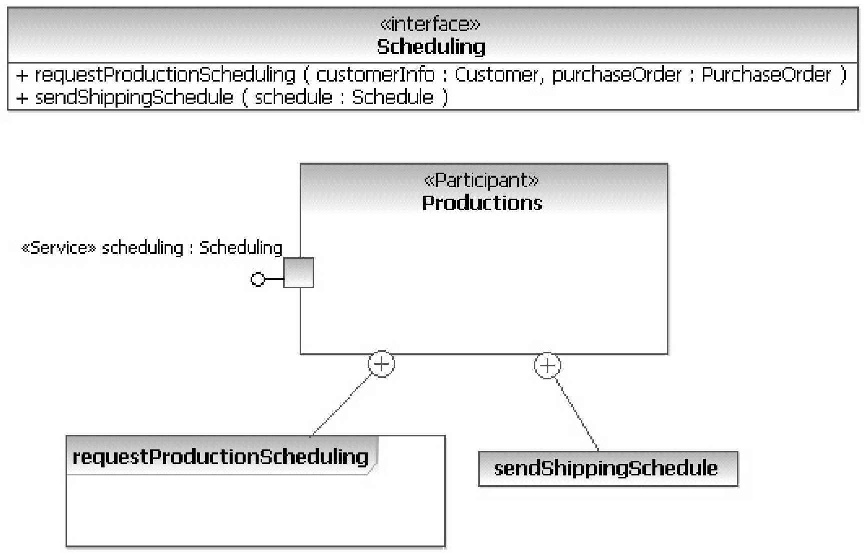
UML does not have this specification and implementation coupling. Each provided operation can have a method behavior that is an activity, interaction, state machine, or opaque behavior (code). The modeler designs the implementation of each operation independently. This can result in situations where the same service provider component uses different behavior types for different operations provided through the same interface. We need some way to translate these service providers to Web services.

There is another factor to consider. In UML, components are instantiated, rather than the behaviors that they own. Therefore, instance identification and lifetime are the same for all behaviors in the same component. In addition, the component establishes a context or scope for all of its owned behaviors, thereby allowing those behaviors to share access to the component state (properties and ports). Thus, when translating to Web services, we must have something that manages this identity, lifetime, and shared state to be able to implement UML semantics. This is where Business Process Execution Language (BPEL) processes come in (see Resources for more about BPEL).

Rather than creating a separate SCA component that implements each behavior of a participant in the module assembly, we create a single BPEL process that corresponds to the component itself. You'll notice that the name of the BPEL process in Figure 13 is **OrderProcessor,** the same as the OrderProcessor service provider, not **processPurchaseOrder**, the name of the provided operation.

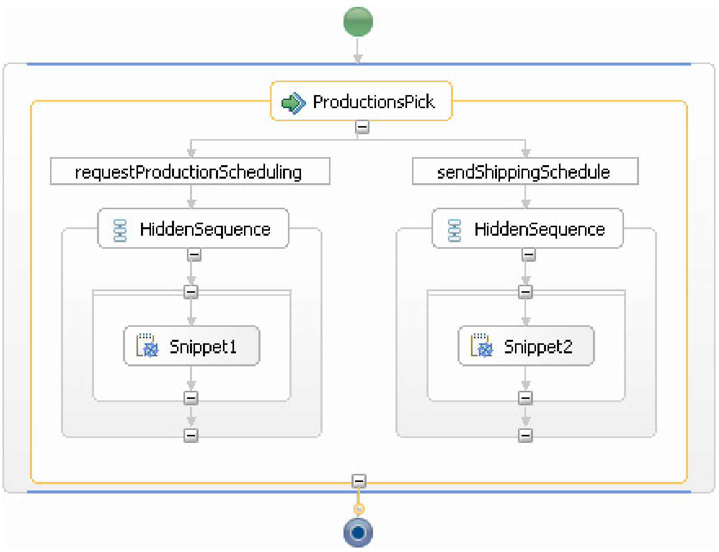
Let's take a closer look at Figure 14. to see how the Productions component is translated to WebSphere Integration Developer Web services.

Figure 14. The Productions Participant


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

Notice that the implementation design for requestProductionScheduling uses an Activity (with the details not shown), but sendShippingSchedule uses an OpaqueBehavior with the implementation provided in Java code. The module assembly for this service provider is shown in Figure 15.

Figure 15. The Productions service provider implementation



A BPEL process is created for the Productions service provider component. The identity properties of the service provider are used to define the correlation for all Invoke, Receive, and Reply activities in this process. Each operation provided by the component is implemented by a fragment of this process. The process starts with a pick element that is used to dispatch each operation request. Then a scope is created for each operation to provide a place for variables defined in the UML behavior. The scope then contains the result of the translated behavior. If the behavior was a UML Activity, then the scope will contain the BPEL generated from that activity. If the behavior was an OpaqueBehavior with Java language, then the body of the behavior is copied into a Java-snipped activity in the scope. If the behavior was an OpaqueBehavior with HTML or JavaServer™ Pages (JSP) language, then a Human Task activity is added to the scope.

This provides complete decoupling of the interfaces and their implementations. For example, if the modeler or developer decided to change a service operation implementation from a human task to an automated Java service, only the human task element in the scope for that operation would need to change. Clients would be unaware that the implementation had changed until they noticed that the service ran faster and they didn't have to do as much work.

Completing the solution

The UML-to-SOA transformation does not generate complete solutions (yet). That's because the effort to put implementation detail in the models would be harder than putting it into the platform-specific artifacts by using editors created for that purpose. There are also some features of UML 2 Activities that are not yet translated to BPEL. These will be added in future releases. Details for what is supported in a particular release are available in Help in Rational Software Architect.

Typically, WebSphere Integration Developer will have to do some or all of the following development activities.

1. **Add the Java code for opaque behaviors** if it was not provided in the model. Even if Java code is provided in the body of the opaque behavior, it is prone to having errors, because the

Content Assist feature and Java (or an action language) compilation are not yet integrated with the modeling capabilities of Rational Software Architect.
2. **Add correlation for business processes.** Correlation specifies information needed to identify instances of a component, and it is not yet supported in the UML-to-SOA transformation. This will be supported in a future release.
3. **Create a user interface (UI) for human tasks.** The UML modeler may have included JSP or HTML in the body of an opaque behavior. But, like Java source code, this may be incomplete or incorrect. The integration developer will want to use the Human Task editor, Page Designer, portal, or other tools to create complete human tasks with the proper look and feel to complete the application UI.
4. **Configure the monitor model.** Currently, the UML-to-SOA transformation does not create any monitor information to evaluate business key performance indicators (KPIs) that may have been modeled as constraints on services and service providers. The integration developer can use the Monitor Model Editor tool in WebSphere Integration Developer to configure what data should be collected for simulation updates and for business measures and metrics.

Series summary

This concludes the series on modeling services in Rational Software Architect by using the IBM Software Services profile (see "More in this series" for the previous four articles). The first article, **Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 1. Service identification**, covered how to use business process models and services architectures to specify the requirements for how a set of services should be connected and choreographed to accomplish some objective. These requirements are often used to specify what has to be accomplished to achieve business objectives. Then the service architectures can be useful in identifying services that provide real business value.

Modeling with SoaML, the Service-Oriented Architecture Modeling Language: Part 2. Service specification showed how to model the details of a service interfaces. A service interface defines information necessary for potential consumers to determine if a service is applicable for their problems, and if so, how to actually use it. A service interface also defines what providers of a service need to do to implement the service.

The next articles, **Part 3. Service realization** and **Part 4. Service composition**, showed how to design and model participants that provide or require services and how the service operations are implemented. They also described how service providers indicate the architectures and service contracts that they fulfill to link the service providers back to the business goals, objectives, processes, requirements and architecture guiding principles.

This last article described how to implement the services on a Web services platform supported by IBM WebSphere Integration Developer by using the IBM UML-to-SOA transformation. WebSphere Integration Developer supports an SOA programming model that is consistent with the service identification, specification, and realization designs captured in Rational Software Architect. By using WebSphere Integration Developer, you can complete the services programming, and then generate a UI for the human tasks, deploy and test your application.

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

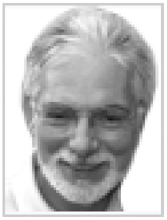
- Download trial versions of other IBM Rational software.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Tivoli®, and WebSphere®.

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.