Using SoaML services architecture

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

Skill Level: Introductory

Date: 24 Apr 2012

The concept of architecture is fundamental to service-oriented architecture (SOA) to encapsulate the interactions between participants. This article explores how to model service architectures by using the OMG SoaML standard.

Introduction

The OMG SoaML specification introduces the concept of services architecture to model how a group of participants that interact through services provided and used to accomplish a result. This is a relatively simple concept that reflects what businesses have been doing for a very long time. But different approaches to modeling in the past, and different options to modeling supported by UML, SoaML, SysML, and UPDM can result in confusion for service-oriented architecture (SOA) modelers.

In this article, you will learn the concepts of services architecture: how the participants are specified, how their interactions are encapsulated and reflected as service agreements, and how to express the results they are intending to deliver. First, you will learn the distinction between class and instance modeling. Hopefully, this will clear confusion that results from practices dictated by past modeling capabilities. Then you will use these concepts to develop a services architecture from two perspectives:

- Top-down design
- Bottom-up abstraction and visualization

Along the way, you will learn the differences between what SoaML refers to as interface-based vs. contract-based approaches to modeling service interactions. This article shows you why these are not actually different ways of modeling the same thing. Instead, they are different modeling capabilities that enable the expression of more interesting and complex interactions.

Then, you will use these different approaches to services modeling in a couple of ways that show the interactions between participants in a services architecture.

Hopefully, this will clarify, demonstrate different modeling options for simple modeling requirements to more complex situations, and help you establish practice guidelines that you can use to get more from your SOA models.

Class modeling compared to instance modeling

Before exploring different kinds of services architecture, it helps to be sure that you understand these two different approaches to modeling things and the connections between things:

Class-based modeling
: is concerned with the description of things that belong to or are classified according to a class. For example, a Person class describes some real-world thing that has a name, lives at an address, and so forth.

Instance-based modeling
: is concerned with specific uses of things that are often, but not always, classified by one or more classes. For example, *Fred* can be an instance of a Person who lives at 220 Irving Lane. The Person class describes the instance of that class, by name.

Some modeling languages, such as UML 1.0 and later and Business Process Modeling Notation (BPMN) 2.0, make limited or no distinction between classes and instances, thus leaving it to the context or the user's needs. For example, Pools in BPMN can represent participants. But there is no way to distinguish between Retail Store as a class and Walmart as an instance. As a result, many modelers used (and still use) UML class diagrams as a means of establishing such a context, where the classes, attributes, and associations depicted in a particular diagram are treated as representing some use of those classes and often described by the class diagram name. The appeal of this approach is that it is simple. You just create a class diagram to show what you are interested in, and you do not have to worry about class vs. instance diagrams nor internal structure.

But there's a high cost associated with this approach. It uses diagrams, or views, to do two things:

- Communicate something to a stakeholder
- Define semantic meaning in the underlying model

However, this coupling of model and view raises real issues about what the diagrams and models mean. For example, do properties, associations, methods, super classes, and so forth shown on one diagram apply to the classes when they appear on other diagrams? Or do you expect tools to use diagrams to define some semantic context and use that to reason about the meaning of the model or as input to transformations? Do connections between classes imply that all instances of those classes must be connected as specified or only the instances in a particular diagram? It's impossible to know, because the meaning is based on convention, not formal modeling techniques. There's no way to distinguish between something that has

been elided away to remove unnecessary detail to aid communication and things that have semantic meaning in one diagram, but not another.

UML 2 provides explicit support for class and instance modeling to avoid this semantic ambiguity. Classes can be used to define specific contexts where parts in their internal structure explicitly model references to instances of other classes in an assembly. The same classes can be used to define other parts in some other context, and these uses are completely independent. This decoupling is fundamental to reuse, which plays an important role in SOA.

SoaML uses UML 2 modeling capabilities to help separate concerns and foster reuse in services architectures. It depends on the clear separation of how something is viewed by different stakeholders from the semantic meaning that applies across all views.

But there's still confusion in UML about connectors in a composite structure vs. the use of dependency wires on class diagrams to define relationships between classes. SoaML does not restrict the use of dependency wires, but SoaML ServicesArchitecture, as described in the specification, generally would not use them. (More on this later in the explanation of the connection between participants in a services architecture.)

Creating a services architecture

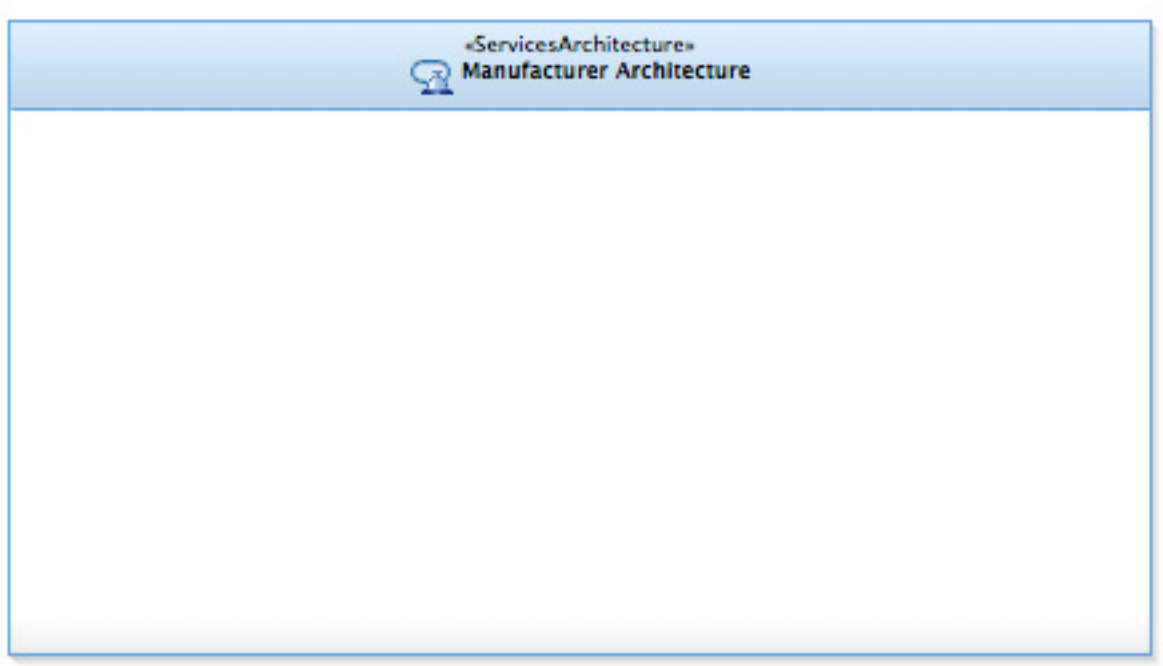
There are two common approaches to defining SoaML ServicesArchitecture:

- The first is top-down design where the services architecture is defined to provide a context for exploring the participants and how they are connected to accomplish a result. This approach is good for discovering candidate services that might need to be adapted or built.
- The second approach is bottom-up, where the services model already exists and the ServicesArchitecture is created as a means of describing that model at a high level.

Although these two approaches are quite different in terms of how the ServicesArchitecture is created and what it is being used for, the contents are the same. This article looks at both cases through the development of a top-down example. Then it shows how to create a bottom-up view of the final result to explore different display options.

When creating a ServicesArchitecture by using the top-down design approach, it is important to think about the problem that is being solved or what you are attempting to accomplish. This should suggest a name for the ServicesArchitecture that is meaningful to potential users. By using the purchase order processing example that's in the SoaML specification as your problem space, and approaching that problem space from the perspective of a manufacturer of products to be sold to consumers, you can define a services architecture for a manufacturer.

Figure 1. The Manufacturer architecture



Add the participants

Next, you add the participants. These are represented as parts or roles in the services architecture, which is an extension of the UML Collaboration. These participants are involved in manufacturing:

Order handler
: Responsible for processing orders

Productions
: Fulfills the order, which possibly involves the development of new or custom products

Shipper
: Ships the order from the productions center to the customer that placed the order

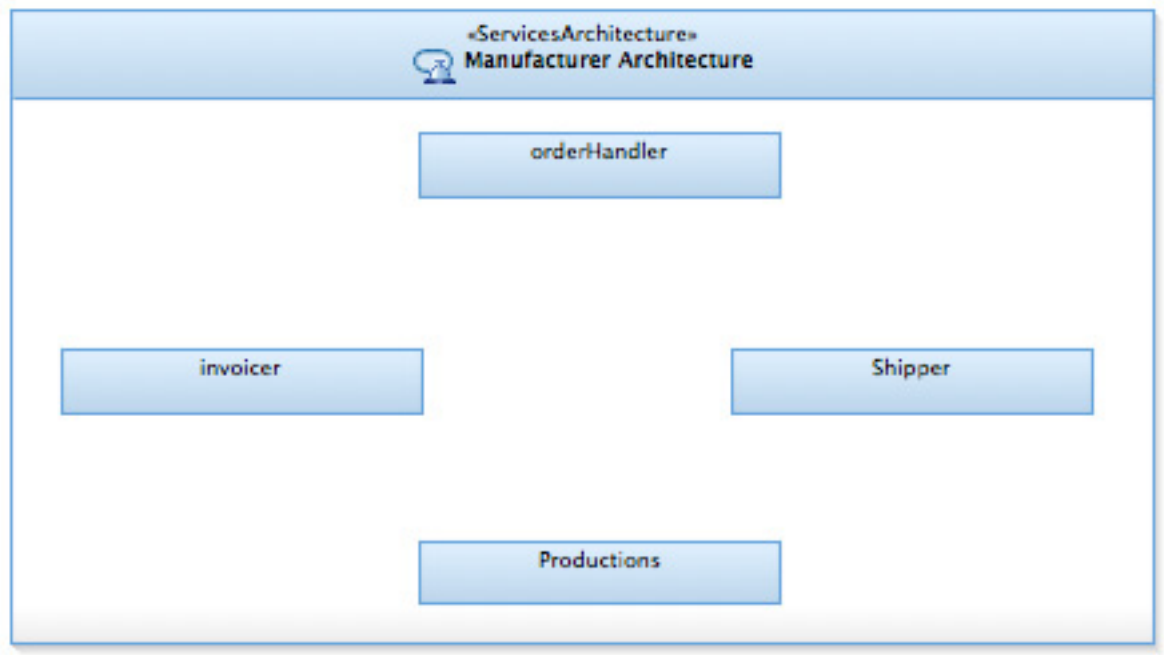
Invoicer
: Processes the invoice for the order and handles all billing functions

This architecture represents a demand-side view of manufacturing, where the products that are produced are heavily influenced by the orders received. Other types of services architectures might take a different view, such as the supply side, and might involve some of the same participants or different ones.

In the top-down scenario, you might or might not have participant classes that define the types for these parts. That does not really matter, because you can easily use existing classes that are available, add the type for the parts or roles later, or create

the participant classes as you add the parts. This is a tool and workflow issue, not something that is specific to UML collaborations or SoaML service architectures. So keep things simple and assume that the participant classes have not yet been defined, so you will create the participants in the services architecture as parts without types. You will name these parts to suggest the role they play in the service architecture.

Figure 2. Adding the participants

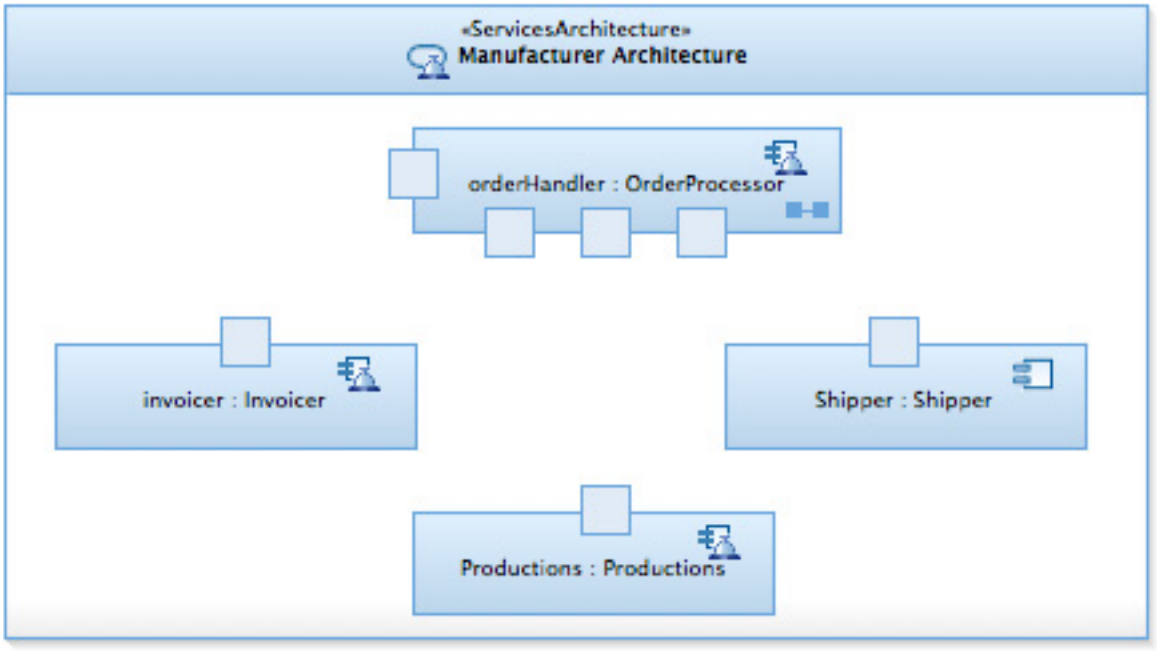


Now, examine the services architecture in the context of your overall enterprise or solutions architecture, perhaps by searching asset reuse libraries to get the participant classes. After those classes are defined, you can use them to set the types of the parts in the services architecture. It is best to do this before you specify the connections between the participants, because these types determine what connections are valid. Of course, you could create connectors between the parts with unspecified types to show which participants need interact, but these connections cannot be validated in any way until the types are specified.

Note:
If the participant types are not specified, it will not be possible to specify the role bindings between the participants and a service contract that represents the agreements between the parties. The participants must be known for these connections to be specified. The next section covers this.

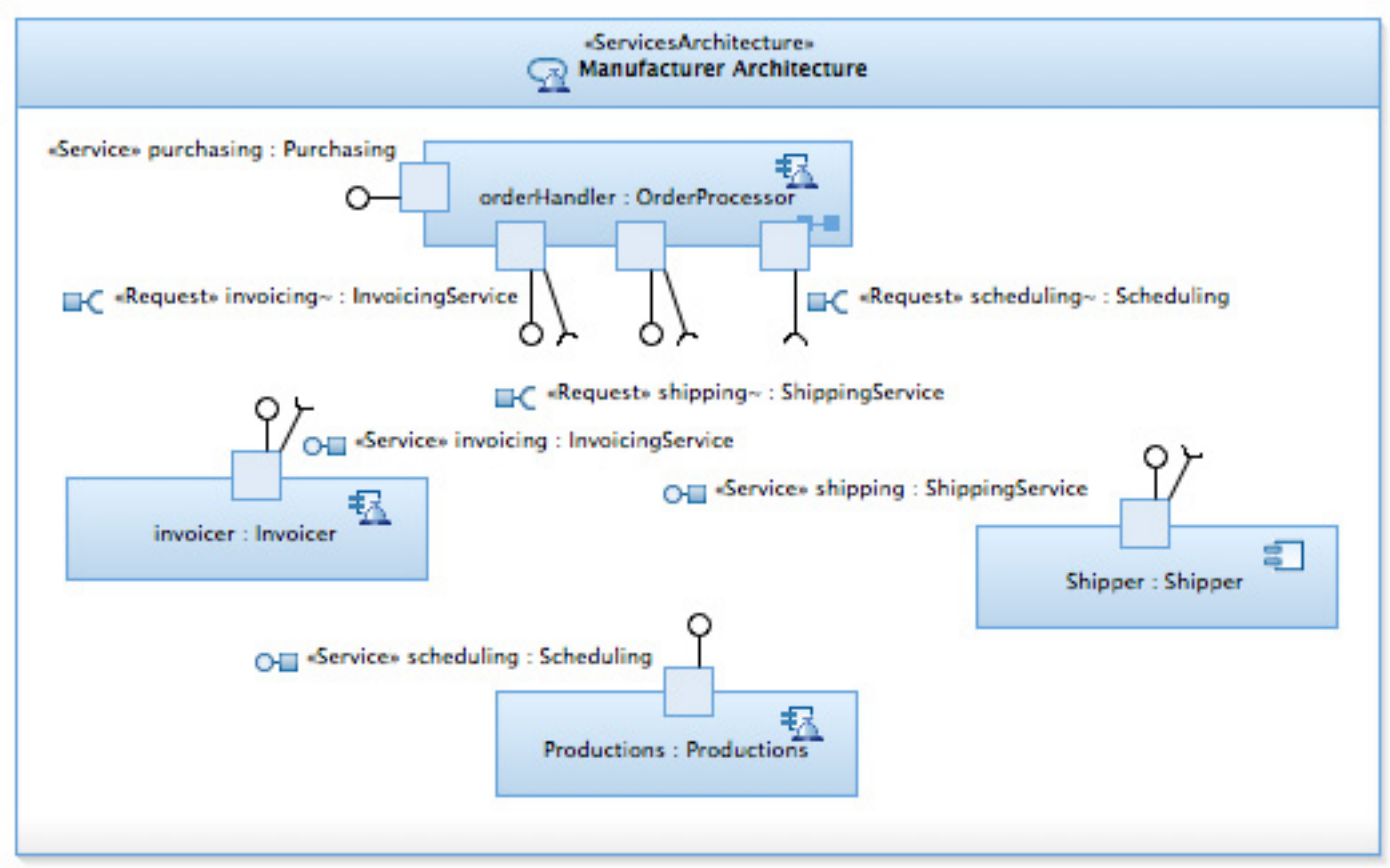
Figure 3 shows the updated services architecture, with the participant types specified for the participant. Notice that the participant names suggest the things that the class describes but the role names suggest the role that those classes play in a particular context.

Figure 3. Setting the participant types



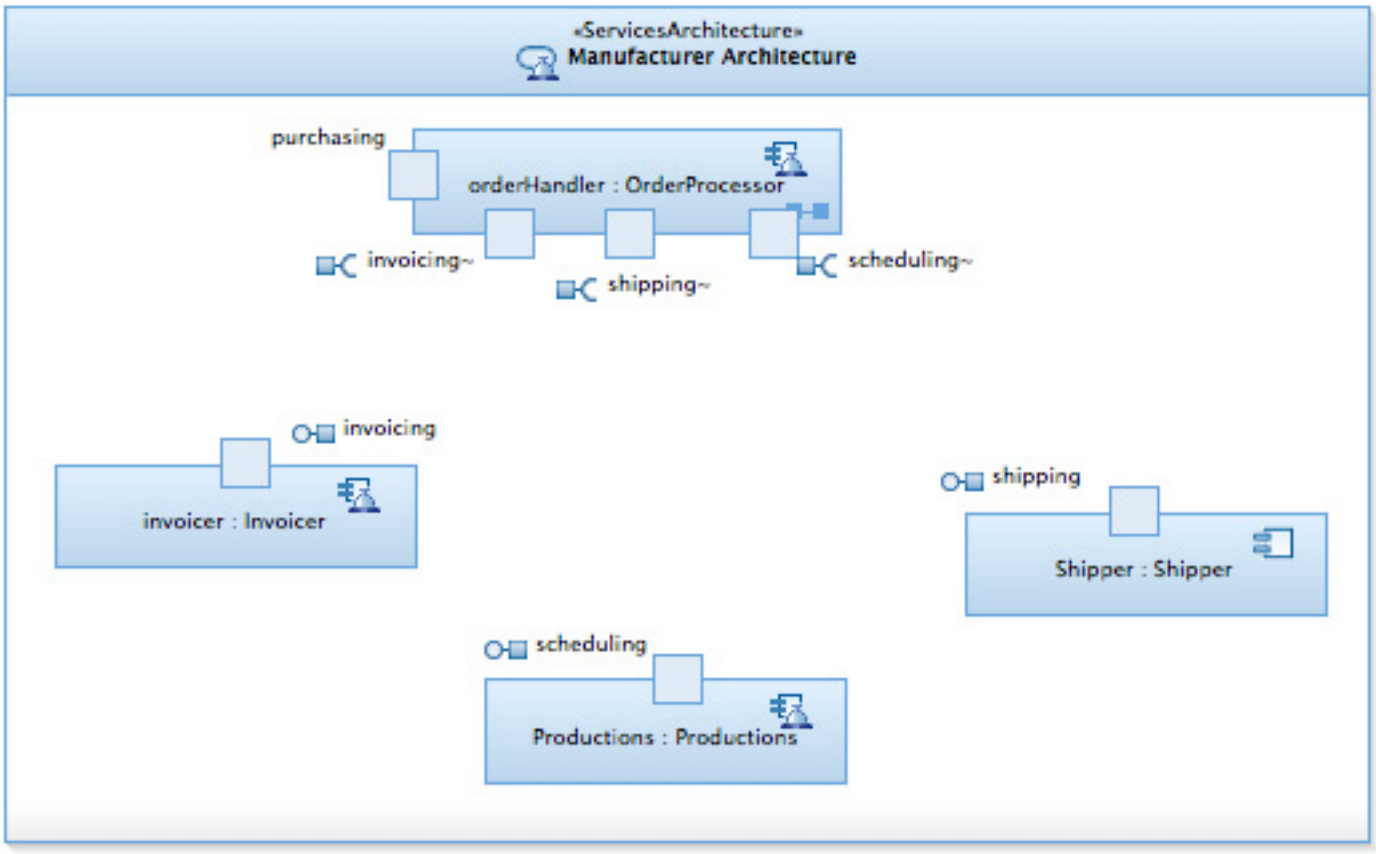
Notice that the roles now show ports that depict the services provided and used for of each participant (if they are defined). This enables you to show additional details about the actual services involved in the services architecture, who provides them, and who uses them. The same diagram can show more detail if you turn on the labels for the ports to see the detailed services and requests, as well as the provided and required interfaces.

Figure 4. Diagram of the service details



If this is too much detail, you can use appearance properties to select a better format for the port labels and typed element label style. Based on the selections, the sample diagram (Figure 5) displays stereotypes as decorations, and the port labels use names only.

Figure 5. Hiding unnecessary details



The point is that having full information in the model does not mean that everything has to be shown all of the time on every diagram. Modeling tools and UML notations give you the flexibility to display only what stakeholders need, while retaining the full information in the model.

You now know the services architecture for manufacturing and what participants are involved. But you don't yet know how the participants interact or what results the services architecture is intended to deliver.

Before expanding on the model to capture this information, consider these different approaches for defining the services that specify potential interactions between the participants.

Define services

This is another area where there is confusion in the SoaML specification. SoaML attempted to provide facilities for defining simple and complex services. The specification talks about two different approaches to defining services:

- Interface-based
- Contract-based descriptions

These are not different ways of saying the same thing; rather, they are different modeling capabilities that provide additional ways of expressing interactions with various degrees of complexity. They can be used together in the same model or even on the same service. For example, it might be useful to define a set of related services by using service interfaces, but then to combine them into a larger multiparty service by using a service contract. Another way to think about it is that a service *interface* defines a service by specifying the constraints on provided and required interfaces. A service *contract* defines a more complex service by specifying the constraints on a set of service interfaces.

In UML, a component encapsulates an implementation of a certain behavior. The interface to the component is defined separately, so users are decoupled from any particular implementation, which allows substitutions and evolution over time without unnecessary impact on clients. UML supports different approaches to defining the interface of a component:

- The component can realize or use any number of interfaces to specify the operations that it provides and those that it requires.
- The component can realize a specification component, which defines its static and dynamic interface.
- The component can provide a number of ports with types that describe the encapsulation of interactions with other components.

These three approaches support different techniques for managing coupling between components. Specification components allow associations and connections between components to be at the specification level, not the implementation. A specification component is richer than a simple interface in that it can specify provided and required interfaces, and expected behavior or component lifecycles all realizing components must adhere to. all independent of any implementation of the specification component. This reduces coupling by eliminating dependencies on particular component implementations. Factory methods can be used to select an implementation that meets the developer's needs.

Ports provide further decoupling for either components or specification components by encapsulating and separating the interactions that a component has with other components. This is the foundation of SOA — managing cohesion and coupling through encapsulation of specification, implementation, and interactions between components.

In most modern programming languages, the component realizes one or more interfaces to define a specification. UML 2 and SoaML go a step further, abstracting specific interaction points as ports where specific connections can be made. This reduces coupling in an SOA by abstracting the interactions between participants on separate ports. Changes in interactions with one participant, or substituting a different participant, do not have any impact on interactions with other participants. The ports

act to decouple the participants. SoaML restricts UML to require that all interactions between participants be through connections to ports.

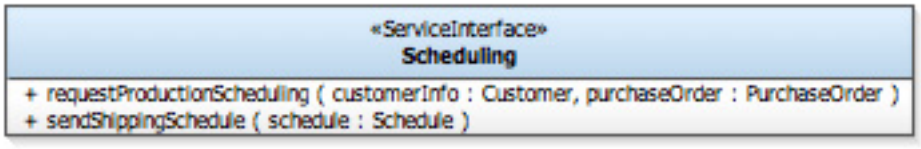
SoaML recognizes that not all services are the same. Some are more complex than others in that they might have both provided and required interfaces, the interactions might need to follow some protocol, and the interactions might involve multiple parties over long periods of time. To support this variability, SoaML provides a several ways of describing a service:

- Simple UML interface to type a service port
- SoaML ServiceInterface to type a service port
- SoaML ServiceContract

Simple UML interface used to type a service port

This interface describes simple interactions between consumers and providers through operations that can be called by consumers and receptions that will be sent to providers. But it cannot express any protocols for when those operations can or should be called, when the reception of events will occur, or in what order. For example, the Scheduling ServiceInterface is just a simple UML 2 interface.

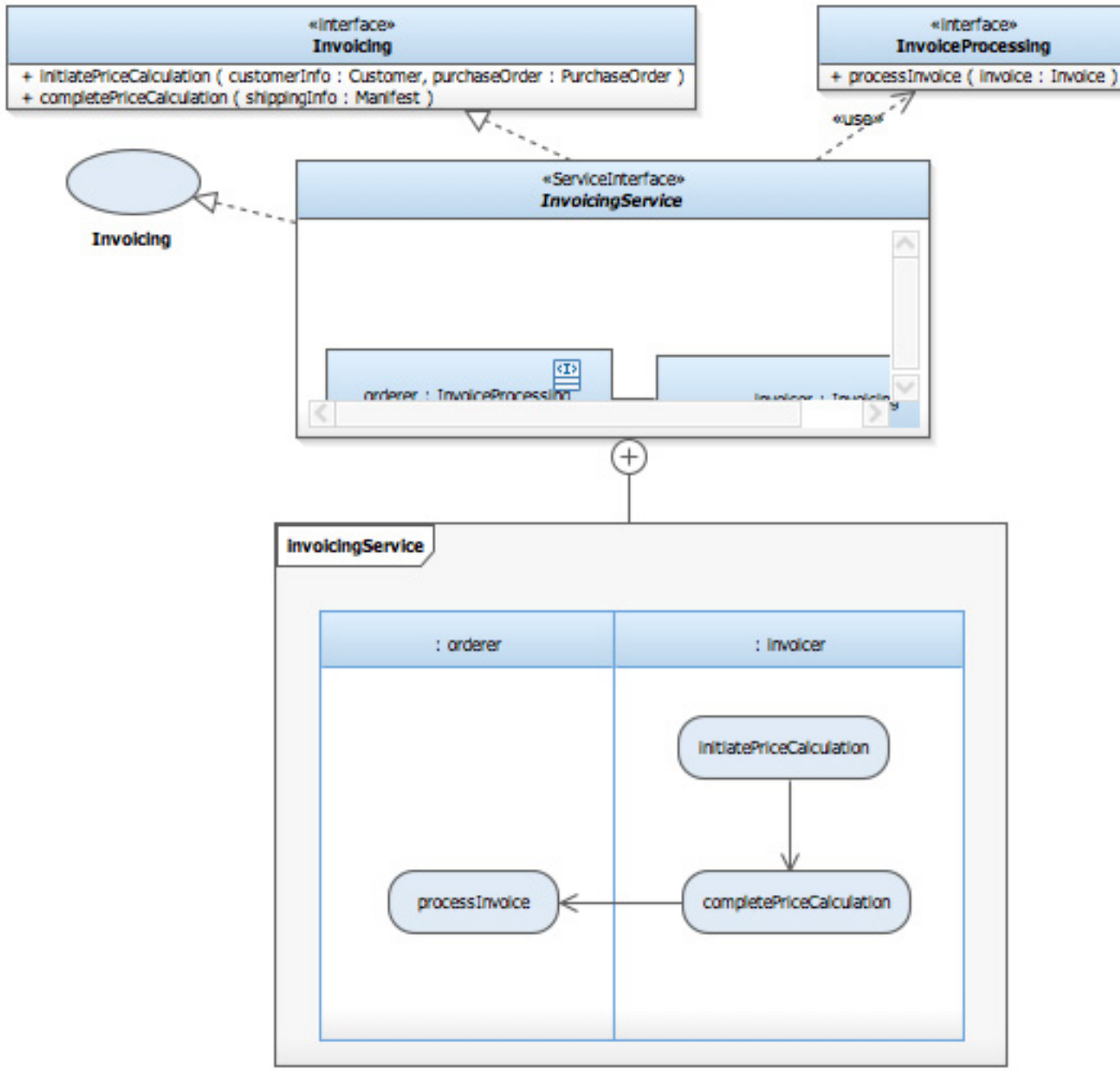
Figure 6. A simple service interface



SoaML ServiceInterface to type a service port

The SoaML ServiceInterface extends the simple Interface with the ability to describe provided and required interfaces and protocols (through the ServiceInterface's owned behaviors). For example, the InvoicingService is a more complex service interface that requires a protocol. The protocol indicates that the computePriceCalculation must be invoked after the initiatePriceCalculation.

Figure 7. A complex service interface with a protocol

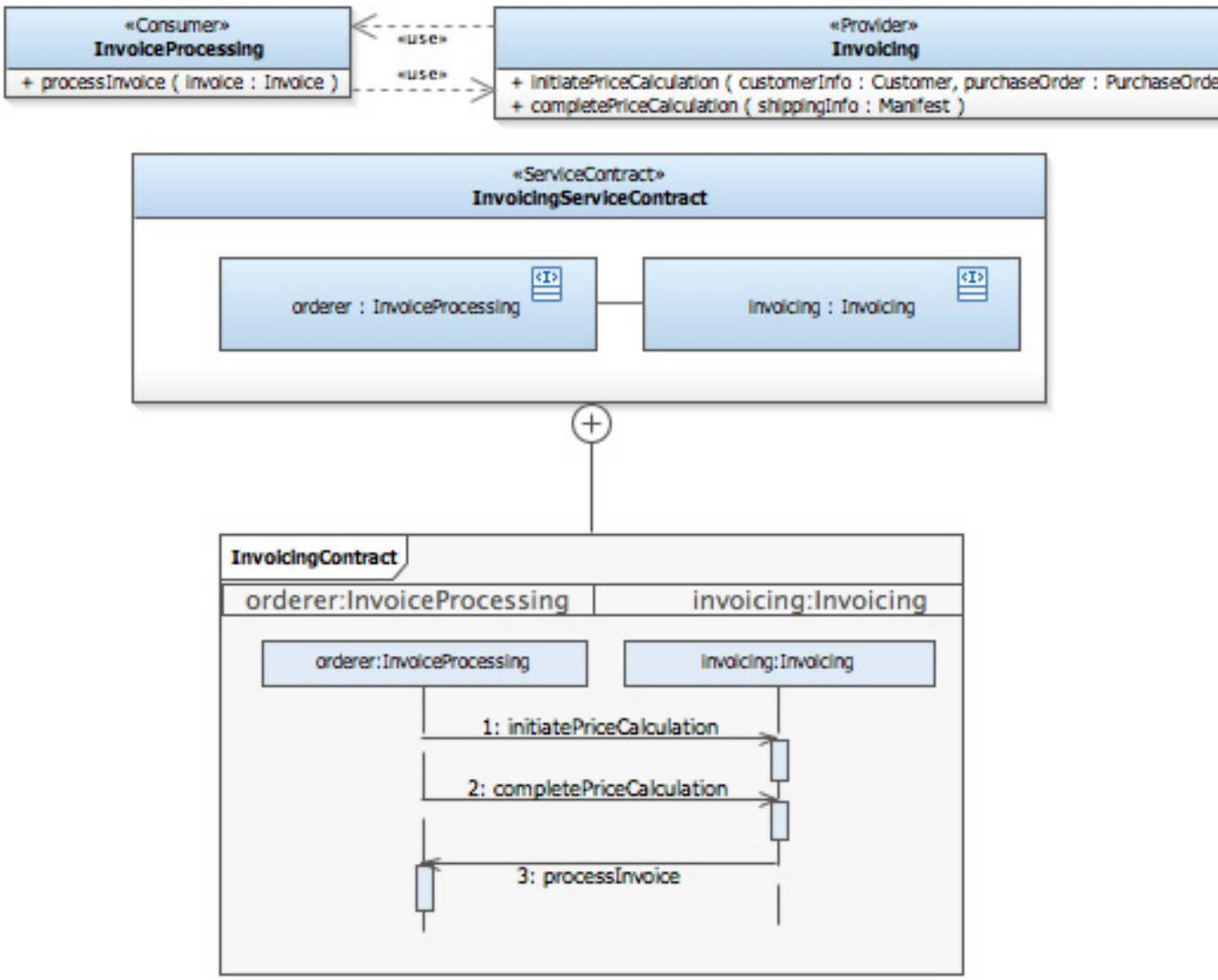


SoaML ServiceContract

This interface defines additional constraints on ports through specification of specific roles that consumers and providers play in the context of a particular service, separated from any particular consumers or providers. Service contracts are very useful for multiparty services.

Figure 8 shows the InvoicingServiceContract.

Figure 8. A service contract



There is some overlap between these approaches, and how to deal with that overlap is a matter of style. At a minimum, each port must have a type — at least an interface. If there is a simple protocol, it can be described in a ServiceInterface that is easily visible to all consumers and defines the value proposition, capabilities, and commitments of the provider. Consumers, in turn, can define Request ports that specify their goals, needs, and expectations. The ServicesArchitecture then connects the service consumer request ports to compatible service provider service ports through simple connectors or SoaML ServiceChannels.

In more complex situations, there might be additional constraints on a particular service or requests that are best described independently from any particular consumer or provider. This is often true in multiparty service interactions. A ServiceContract is useful in this case. A ServiceContract is an extension of the UML

Collaboration and provides a means of specifying the interactions between a set of roles that participants are expected to play when they engage in a service defined by that service contract. This collaboration defines the agreement between the participants, independent of any particular participant, thereby providing even more decoupling between the individual participants. A services architecture can show the agreements and connections between participants that are using service contracts by binding the participant parts in the services architecture to the roles that they play in the service contract. These participants are then constrained to provide ports that are compatible with the type of the roles they are bound to and to behave according to the specification of the service contract when interacting through that port. The type of the port must be compatible with the role that it is bound to, but it does not need to be exactly the same type. And the type of the port could be a simple UML interface or a more complex SoaML ServiceInterface.

Therefore, there really are not two different ways of describing services in SoaML, interfaced-based and contract-based. Rather, there are ways of defining services with different modeling capabilities to address different levels of complexity. You could use any one of the three approaches to model a wide range of service description requirements:

- UML 2 interfaces can be associated with protocol state machines that can describe service or request protocols for using the operations and receptions defined in the interface.
- A SoaML ServicesInterface can also have multiple parts to constrain interactions between more than two participants.
- A SoaML ServiceContract can be used to model a service that involves two roles, one of which is not explicitly specified.

But in practice, the best approach, as previously outlined here, is to use simple interfaces when they are sufficient, use service interfaces when protocols are required, and use service contracts for multiparty interactions.

Connect the participants

You now have the services interfaces and contracts defined, the services architecture has been created, and you have added the participants. Next, you address how the participants are connected.

Recall that the hallmark feature of SOA is that the interactions between parties are abstracted and encapsulated to reduce and manage the coupling and dependencies between service consumers and providers. This not only encourages and enables reuse of services and service providers; it also enables service consumer flexibility in choosing which service providers to use in their value chain, without having undue impact across the providers. It is these connections between participants in a services architecture that is a fundamental part of the SOA and that establish the coupling in the system that must be managed.

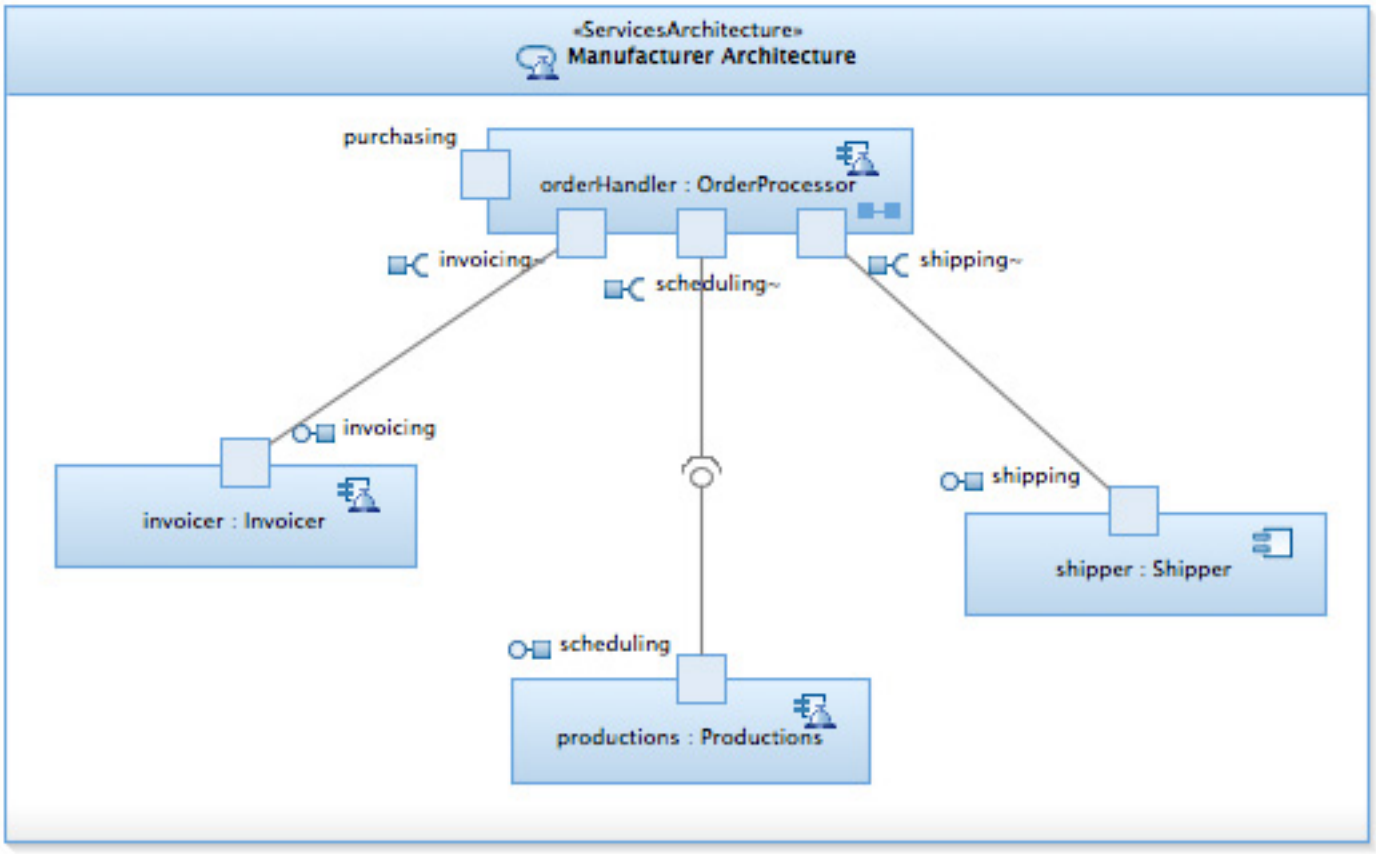
Also, recall that you are connecting instances of participants in a services architecture for a particular purpose. You are not constraining all instances of those participants to be connected the same way. Different architectures might use the same or different participant types or use services in different ways. This is exactly what you need to support reuse and flexible business integration.

There are choices for how to connect participant parts, depending on the complexity of the interactions and the level of detail that is to be captured in the services architecture. The interaction between the participants can be shown in three ways.

Using ServiceChannels
These can be between the UML parts (properties) if the ports are not yet defined, or they can be between service and request ports. This would be typical of services that are defined with simple Interfaces or ServiceInterfaces. Figure 9 shows the participants in the Manufacturer architecture connected by service channels. The interactions that occur across the service channels must be consistent with the interfaces and services interfaces that define the connected service and request ports.

Notice that changing the participant that provides the shipping service would have its impact isolated to the shipping request port of the orderHandler participant. Interactions with other participants are isolated by the service and request ports and do not result in direct dependencies on the orderHandler.

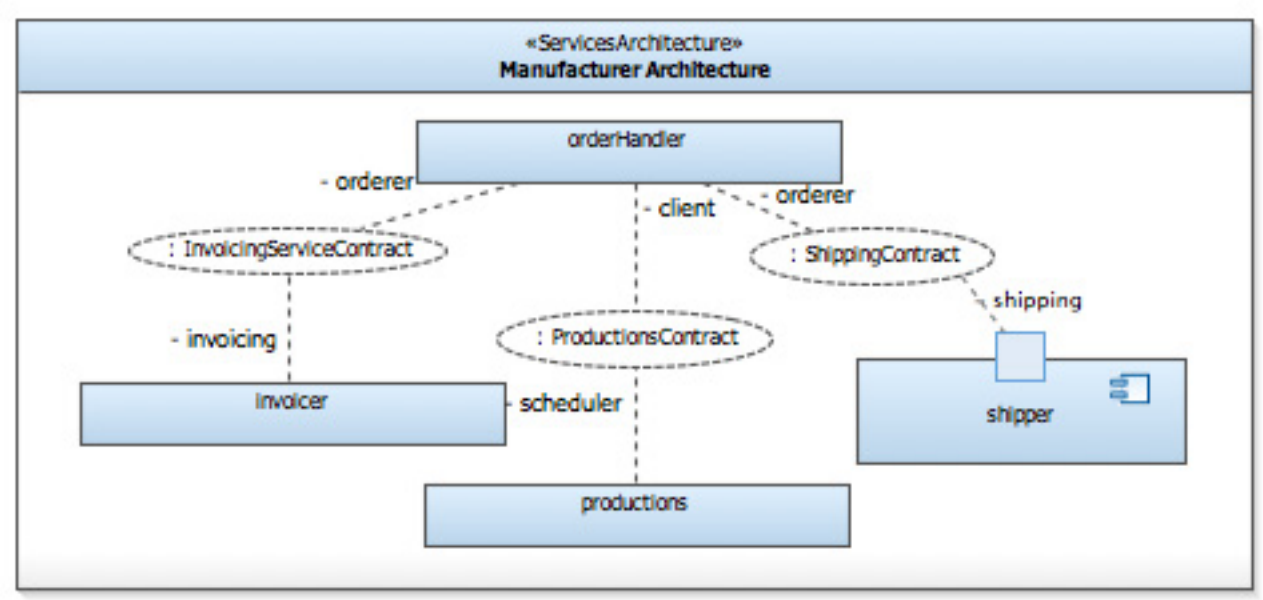
Figure 9. Connecting the participants with connectors



Using ServiceContract bindings

These constrain the participants to the roles that they play in a ServiceContract, as specified by the bindings. Again, the bindings can be to the participant or to a port of the participant if it is known. Figure 10 shows instances of the service contracts for invoicing, productions, and shipping (collaboration uses), with role bindings indicating what part the participants play in the services architecture. This example shows the interactions at a high level, between the participants, with the exception of the shipping role, which is to a particular service request port of the shipper participant. It is this isolation that is a key part of SOA.

Figure 10. Connecting the participants with service contracts



This diagram means the same thing as the previous diagram, but it can model additional constraints on the interactions between the participants.

Using dependency wires

This is supported in UML 2, but it is not used by SoaML to describe services architectures. It might be useful for quick, informal sketches of the intended interaction between representative instances depicted by class diagrams.

As already mentioned, SoaML does not recommend the use of dependency wires at the class level to indicate interactions between participants, because this leads to greater coupling and less reuse. However, there might be situations where you do want to constrain all instances of participant classes to interact in exactly the same way. Dependency wires would be an appropriate way of expressing this constraint.

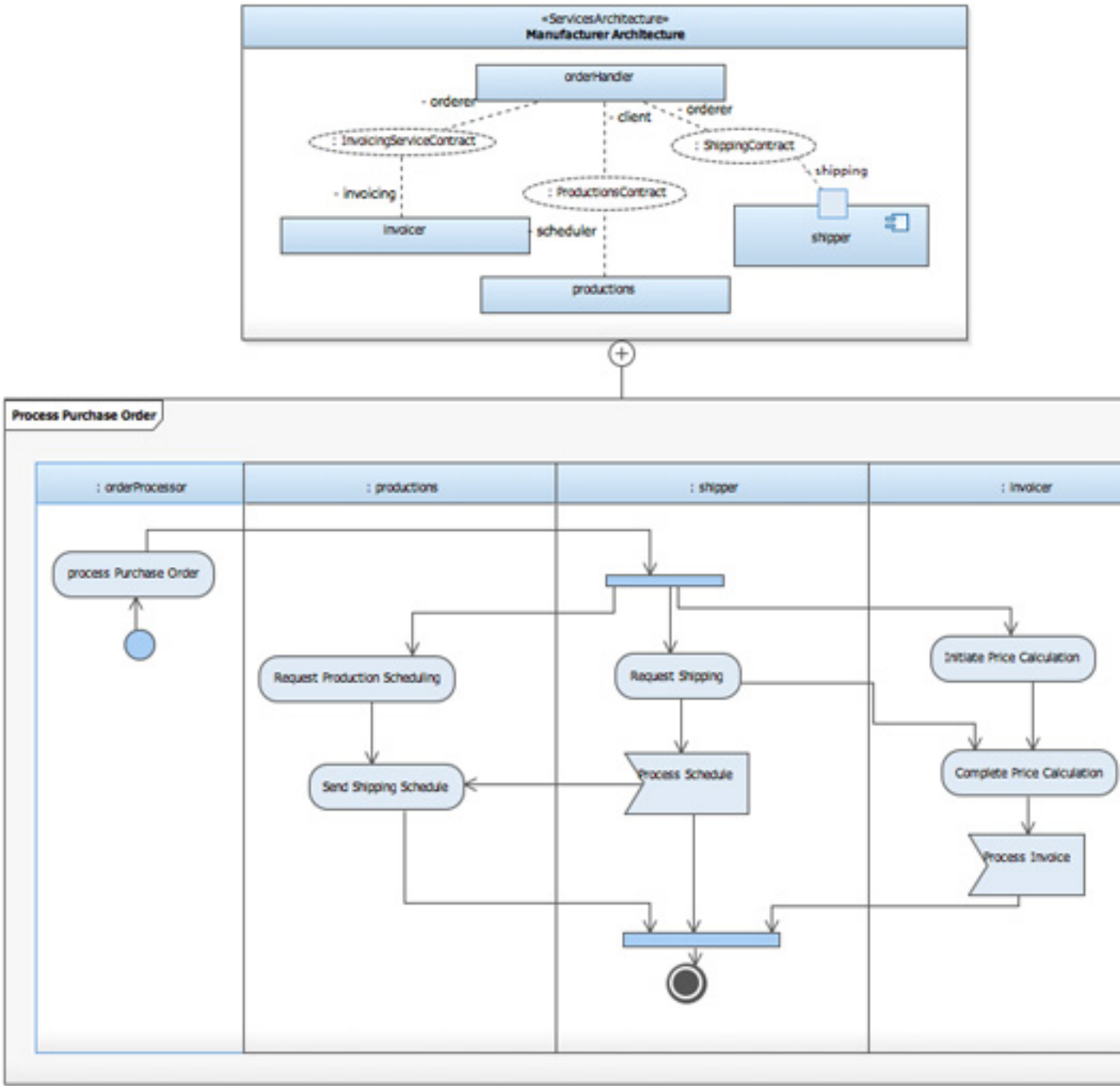
Using service channels or service contracts to model interactions between participant parts in a services architecture mean the same thing. Service contracts provide greater expressivity and can easily model multiparty interactions. The choice is one of style or need for expressivity.

Describe the expected results

What the services architecture is intended to accomplish can be described in its owned behavior. A services architecture is similar to a service contract. A service *contract* specifies how a set of roles (defined by interfaces or service interfaces) work together to achieve the real-world effect of a single service. A services *architecture* specifies how a set of participants interact through a set of services to accomplish some result. In both cases, the expected result can be modeled by a UML 2 behavior: an interaction, activity, state machine, or opaque behavior (defined in a

specified language). For your Manufacturer architecture, the intended real-world effect is to process a purchase order by fulfilling the order, shipping the products, and processing an invoice. A UML activity can be used to express this business logic.

Figure 11. The services architecture business logic



Larger view of Figure 11.

The business logic of the services architecture is an activity owned by the services architecture as indicated by the “circle-plus” notation. The activity partitions represent

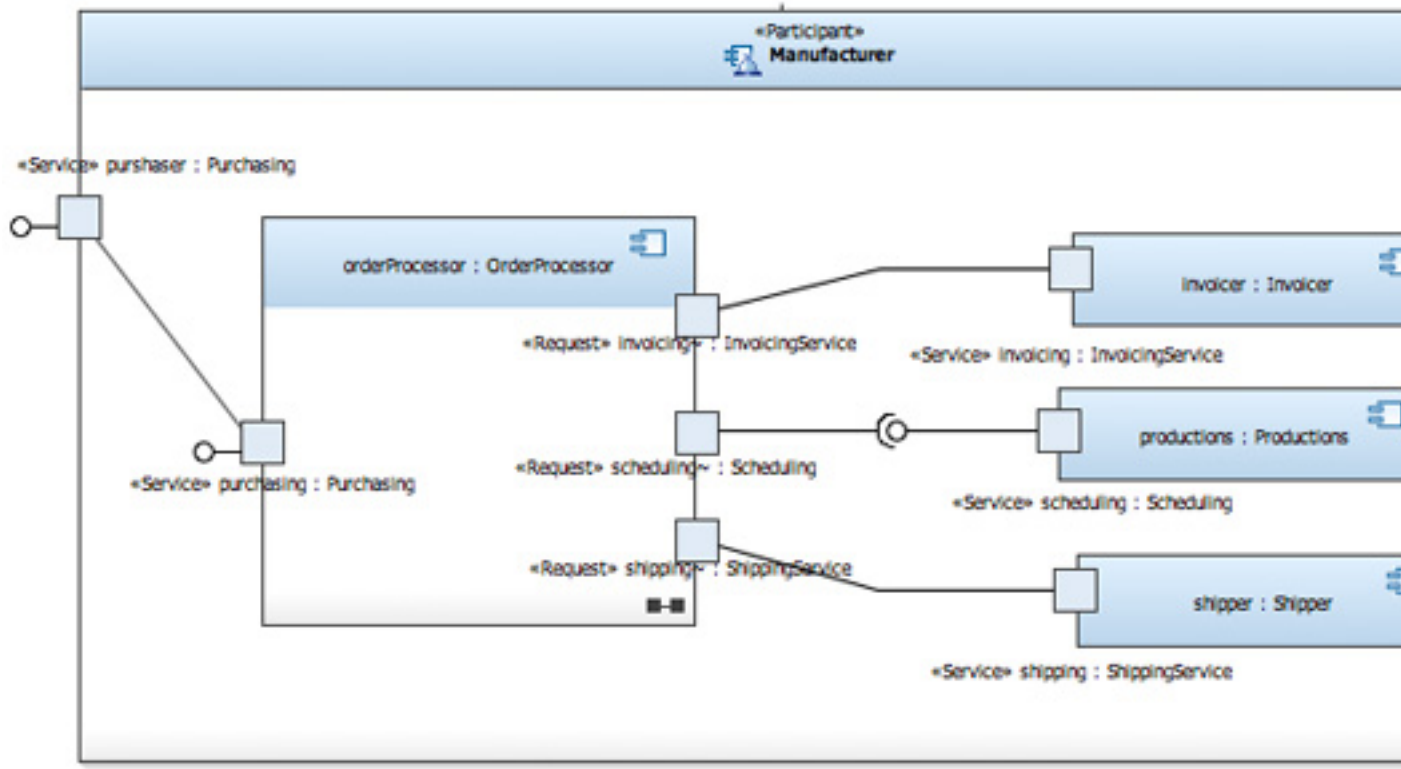
the parts in the services architecture. Actions in each partition indicate what part is responsible for performing that action and when. This activity model is very similar to a BPMN 2.0 Orchestration.

What this behavior means is that the interactions between the service consumers and providers, the parts in the services architecture, as described by the service contracts or connections that model the interactions, must occur in a manner that is consistent with the behavior of the services architecture. This behavior can be sketched when the services architecture is first created. Then the actions in the activity can be used to identify candidate services, which are then specified by the services interfaces or service contracts and realized by the providing participants.

Viewing a services architecture

Now, look at the problem from the perspective of a stakeholder who wants to understand an existing implementation of an Order Processor participant. What such a stakeholder might need is to construct a services architecture that describes the architecture of the interactions between the participants in a particular implementation. For example, consider the implementation of a Manufacturer participant that assembles participants to implement processing purchase orders.

Figure 12. Assembling participants



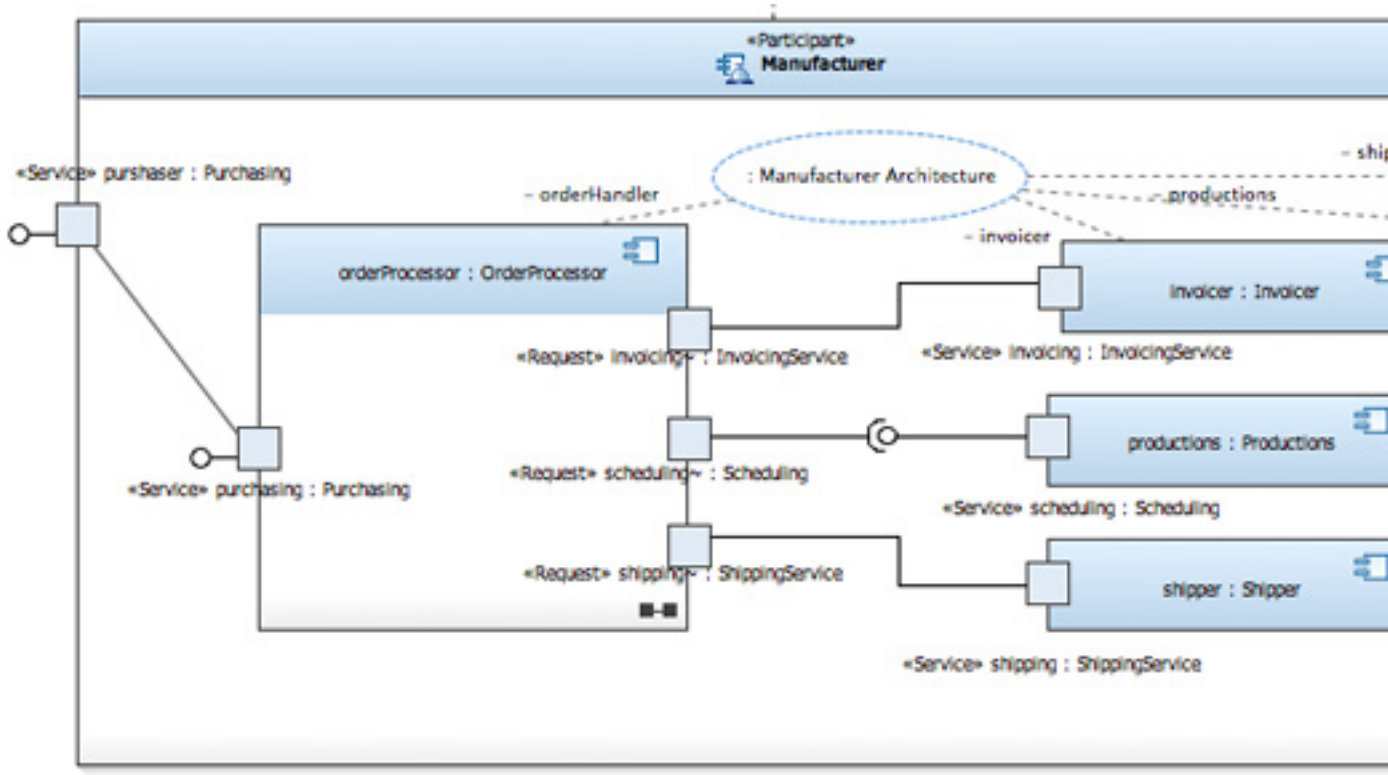
Larger view of Figure 12.

This implementation shows a number of participants connected through service channels that will behave a certain way given some request. What you want to do is to construct the services architecture that describes or constrains this interaction. You can do this by creating a ServicesArchitecture, and dragging and dropping the participant classes into the services architecture to create participant parts with the appropriate type. You can then drag the services contracts into the diagram to create uses of those contracts and bind the parts to the appropriate roles. This builds up the services architecture from the already implemented participants, classes, service contracts, and service interfaces, jus by using the simple drag-and-drop technique. Modelers can then adjust the appearance and options to display the information that meets their needs.

The services architecture created would be exactly the same as the one that you created by using top-down design. Tools can provide different display options to elide away unnecessary detail so that you can provide the same simple, high-level diagrams that you might have started with to sketch the original services architecture design.

And finally, just as you can show what roles participants play in a single service with service contracts between the participants in a services architecture, you can show how the participants in an implementation adhere to a services architecture. In this case, you bind the parts in the implementation to the roles that they play in the services architecture.

Figure 13. Participants adhering to a services architecture



Larger view of Figure 13.

Summary

This article covered a lot of ground, including several potentially confusing concepts in UML and SoaML:

- The difference between class- and instance-based modeling and the concept of class diagrams that depict *representative instances* as opposed to *composite structure diagrams* that depict references to actual instances as a means of separating the specification of things from their uses in a particular context
- Creating a services architecture as a means of top-down service discovery and design, including specifying the services architecture, the collaborating participants, the agreements that model and constrain their interactions, and what you intend the architecture to accomplish
- The different ways of specifying simple to complex service descriptions by using UML Interface, SoaML ServiceInterface, or SoaML ServiceContract, or combinations
- Creating a view of a services architecture from an existing implementation and how an implementation can be constrained by a services architecture

I hope that you found this useful and that it cleared some of the confusion that can result from different modeling capabilities that you can use to express similar things.

In a future article, I will cover how SoaML and BPMN 2.0 relate to each other, using SoaML to help understand BPMN 2.0 orchestration, conversation, and choreography, and how you can use the notations together.

Happy modeling!

Resources

Learn

- Find out more about SoaML and SOA:
  - Modeling with SoaML the Service-Oriented Architecture Modeling Language: Part 1. Service identification, by Jim Amsden, is the first in a series of four articles about developing software based on SoaML. (IBM® developerWorks®, January 2010).
  - 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 services 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.
- Visit the Rational software area on developerWorks for technical resources and best practices for Rational Software Delivery Platform products.
- Subscribe to the developerWorks weekly email newsletter, and choose the topics to follow.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools, as well as IT industry trends.
- Watch developerWorks on-demand demos, ranging from product installation and setup demos for beginners to advanced functionality for experienced developers.
- Improve your skills. Check the Rational training and certification catalog, which includes many types of courses on a wide range of topics. You can take some of them anywhere, any time, and many of the "Getting Started" ones are free.

Get products and technologies

- Download a free trial version of Rational software.
- Evaluate other IBM software in the way that suits you best: Download it for a trial, try it online, use it in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement service-oriented architecture efficiently.

Discuss

- Join the Rational software forums to ask questions and participate in discussions.
- Ask and answer questions and increase your expertise when you get involved in the Rational forums, cafés, and wikis.
- Rate or review Rational software. It's quick and easy.

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.