

Service oriented architecture Modelling Language (SoaML™)

UML Profile Reference 1.0

Copyright

Copyright © 2009, Data Access Technologies, Inc. for ModelDriven.org. All rights reserved worldwide. Portions Copyright © International Business Machines, Sintef, Adaptive, Capgemini, EDS, Fujitsu, Fundacion European Software Institute, Hewlett-Packard, Mega International, Rysome, Softeam and the Object Management Group.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBA services™, CORBA facilities™, CORBA med™, CORBA net™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

Table of Contents

Overview of this Document.....	3
SoaML UML Profile Specification.....	4
EXECUTIVE OVERVIEW	4
<i>An architectural and business focused approach to SOA</i>	<i>4</i>
<i>Top down and bottom-up SOA</i>	<i>5</i>
<i>Key Concepts of Basic Services</i>	<i>5</i>
<i>Service Interfaces.....</i>	<i>10</i>
<i>Key Concepts of the Services Architecture</i>	<i>9</i>
<i>Capability.....</i>	<i>10</i>
THE SOAML PROFILE OF UML	18
STEREOTYPE DESCRIPTIONS	19
<i>Agent</i>	<i>19</i>
<i>Attachment</i>	<i>21</i>
<i>Capability.....</i>	<i>22</i>
<i>Consumer</i>	<i>23</i>
<i>CollaborationUse.....</i>	<i>25</i>
<i>Port</i>	<i>27</i>
<i>MessageType.....</i>	<i>28</i>
<i>Milestone.....</i>	<i>31</i>
<i>Participant</i>	<i>32</i>
<i>ParticipantArchitecture</i>	<i>36</i>
<i>Property</i>	<i>38</i>
<i>Provider</i>	<i>40</i>
<i>RequestPoint</i>	<i>41</i>
<i>ServicePoint</i>	<i>43</i>
<i>ServiceChannel</i>	<i>46</i>
<i>ServiceContract.....</i>	<i>48</i>
<i>ServiceInterface</i>	<i>53</i>
<i>ServicesArchitecture</i>	<i>61</i>

Overview of this Document

This is a reference document for the SoaML Profile for the open source ModelDriven.org ModelPro project. The SoaML (Service oriented architecture Modeling Language) specification is an OMG® standard for the design of services within a service-oriented architecture. This document attempts to “track” the SoaML specification as it is expected to emerge from the OMG Finalization Task Force but some changes may be made in such finalization.

The goals of SoaML are to support the activities of service modeling and design and to fit into an overall model-driven development approach. Of course, there are many ways to approach the problems of service design. Should it be taken from the perspective of a service consumer who requests that a service be built? Should it be taken from the perspective of a service provider that advertises a service to those who are interested and qualified to use it? Or, should it be taken from the perspective of a system design that describes how consumers and providers will interact to achieve overall objectives? Rather than presume any particular method, the profile accommodates all of these different perspectives in a consistent and cohesive approach to describing consumers requirements, providers offerings and the interaction and agreements between them.

The SoaML profile supports the range of modeling requirements for service-oriented architectures, including the specification of systems of services, the specification of individual service interfaces, and the specification of service implementations. This is done in such a way as to support the automatic generation of derived artifacts following an MDA based approach.

Additional, related documentation may be found on <http://www.ModelDriven.org/ModelPro>. Of particular interest will be the specification of the provisioning and JEE Cartridge profiles that work with SoaML. All ModelPro profiles are included in the “Model Driven Profile”.

SoaML UML Profile Specification

Executive Overview

Service Oriented Architecture (SOA) is a way of organizing and understanding organizations, communities and systems to maximize agility, scale and interoperability. The SOA approach is simple – people, organizations and systems provide services to each other. These services allow us to get something done without doing it ourselves or even without knowing how to do it—enabling us to be more efficient and agile. Services also enable us to offer our capabilities to others in exchange for some value – thus establishing a community, process or marketplace. The SOA paradigm works equally well for integrating existing capabilities as well as creating and integrating new capabilities.

A service is an offer of value to another through a well-defined interface and available to a community (which may be the general public). A service results in work provided to one by another.

SOA, then, is an architectural paradigm for defining how people, organizations and systems provide and use services to achieve results. SoaML as described in this specification provides a standard way to architect and model SOA solutions using the Unified Modeling Language® (UML®). The profile uses the built-in extension mechanisms of UML to define SOA concepts in terms of existing UML concepts. SoaML can be used with current “off the shelf” UML tools but some tools may offer enhanced, SOA specific.

An architectural and business focused approach to SOA

SOA has been associated with a variety of approaches and technologies. The view expressed in this specification is that SOA is foremost an approach to systems *architecture*, where architecture is a way to understand and specify how things can best work together to meet a set of goals and objectives. Systems, in this context, include organizations, communities, processes as well as information technology systems. The architectures described with SOA may be business architectures, mission architectures, community architectures or information technology systems architectures – all can be equally service oriented. The SOA approach to architecture helps with *separating the concerns* of *what* needs to get done from *how* it gets done, *where* it gets done or *who or what* does it. Some other views of SOA and “Web Services” are very technology focused and deal with the “bits and bytes” of distributed computing. These technology concerns are important and embraced, but are not the only focus of SOA as expressed by SoaML.

SoaML embraces and exploits technology as a means to an end but is not limited to technology architecture. In fact, the highest leverage of employing SOA comes from understanding a community, process or enterprise as a set of interrelated services and then supporting that *service oriented enterprise* with *service-enabled systems*. SoaML enables business oriented and systems oriented services architectures to mutually and collaboratively support the enterprise mission.

SoaML depends on Model Driven Architecture® (MDA®¹) to help map business and systems architectures, the design of the enterprise, to the technologies that support SOA, like web services and CORBA®. Using MDA helps our architectures to outlive the technology of the day and support the evolution of our enterprises over the long term. MDA helps with *separating the concerns* of the business or systems *architecture* from the *implementation and technology*.

¹ “Model Driven Architecture”, “MDA”. “CORBA”, “Unified Modeling Language”, “UML” and “OMG” are registered trademarks of the Object Management Group, Inc.

Top down and bottom-up SOA

SoaML can be used for basic “context independent services”. Such as common Web-Service examples like “Get stock quote” or “get time”. Basic services focus on the specification of a single service without regard for its context or dependencies. Since a basic service is context independent it can be simpler and more appropriate for “bottom up” definition of services.

SoaML can also be used “in the large” where we are enabling an organization or community to work more effectively using an inter-related set of services. Such services are executed in the context of this enterprise, process or community and so depend on the services architecture of that community. A SoaML services architecture shows how multiple participants work together, providing and using services to enable business goals or processes.

In either case, technology services may be identified, specified, implemented and realized in some execution environment. There are a variety of approaches for identifying services that are supported by SoaML. These different approaches are intended to support the variability seen in the marketplace. Services may be identified by:

Designing services architectures that specify a community of interacting participants, and the service contracts that reflect the agreements for how they intend to interact in order to achieve some common purpose

Organizing individual functions into service capabilities arranged in a hierarchy showing anticipated usage dependencies.

Using a business process to identify functional capabilities needed to accomplish some purpose as well as the roles played by participants. Processes and services are different views of the same system – one focusing on how and why parties interact to provide each other with products and services and the other focusing on what activities parties perform to provide and use those services.

Regardless of how services are identified, their specification includes service interfaces. A service interface defines any interaction or communication protocol for how to properly use and implement a service. A service interfaces may define the complete interface for a service from its own perspective, irrespective of any consumer request it might be connected to. Alternatively, the agreement between a consumer request and provider service may be captured in a common service contract defined in one place, and constraining both the consumer’s request service interface and the provider’s service interface.

Services are provided by participants who are responsible for implementing and using the services. Services implementations may be specified by methods that are owned behaviors of the participants expressed using interactions, activities, state machines, or opaque behaviors. Participants may also delegate service implementations to parts in their internal structure which represent an assembly of other service participants connected together to provide a complete solutions, perhaps specified by, and realizing a services architecture.

Services may be realized by participant implementations that can run in some manual or automated execution environment. SoaML relies on OMG MDA techniques to separate the logical implementation of a service from its possible physical realizations on various platforms. This separation of concerns both keeps the services models simpler and more resilient to changes in underlying platform and execution environments. Using MDA in this way, SoaML architectures can support a variety of technology implementations and tool support can help automate these technology mappings.

Key Concepts of Basic Services

A key concept is, of course, service. Service is defined as an offer of value to another party, enabled by one or more capabilities. Here, the access to the service is provided using a prescribed contract and is exercised consistent with constraints and policies as specified by the service contract. A service is provided by a participant acting as the *provider* of the service—for use by others. The eventual *consumers* of the service may not be known to the service provider and may demonstrate uses of the service beyond the scope originally conceived by the provider. [OASIS RM]

Participants providing and consuming services

A participant is a provider and/or user of services in some context. Participants may be people, organizations, technology components or systems. Participants offer services through “Ports” via the «ServicePoint» stereotype and request services on Ports with the «RequestPoint». These ports represent points where the service is offered or consumed by the participant. *A participant can provide and use any number of services.* Participants that are implemented in information systems are typically SOA enabled components that plug into an application server.

The service point has a type that describes how to use that service, that type may be either a UML interface (for very simple services) or a ServiceInterface. In either case the type of the service point specifies, directly or indirectly, everything that is needed to interact with that service – it is the contract between the providers and users of that service.

Figure 1 depicts a “Productions” participant providing a “scheduling” service. The type of the service port is the UML interface “Scheduling” that has two operations, “requestProductionScheduling” and “sendShippingSchedule”. The interface defines how a consumer of a Scheduling service must interact whereas the service point specifies that participant “Productions” has the capability to offer that service – which could, for example, populate a UDDI repository. Note that a participant may also offer other services on other service points. Participant “Productions” has two owned behaviors that are the methods of the operations provided through the scheduling service.

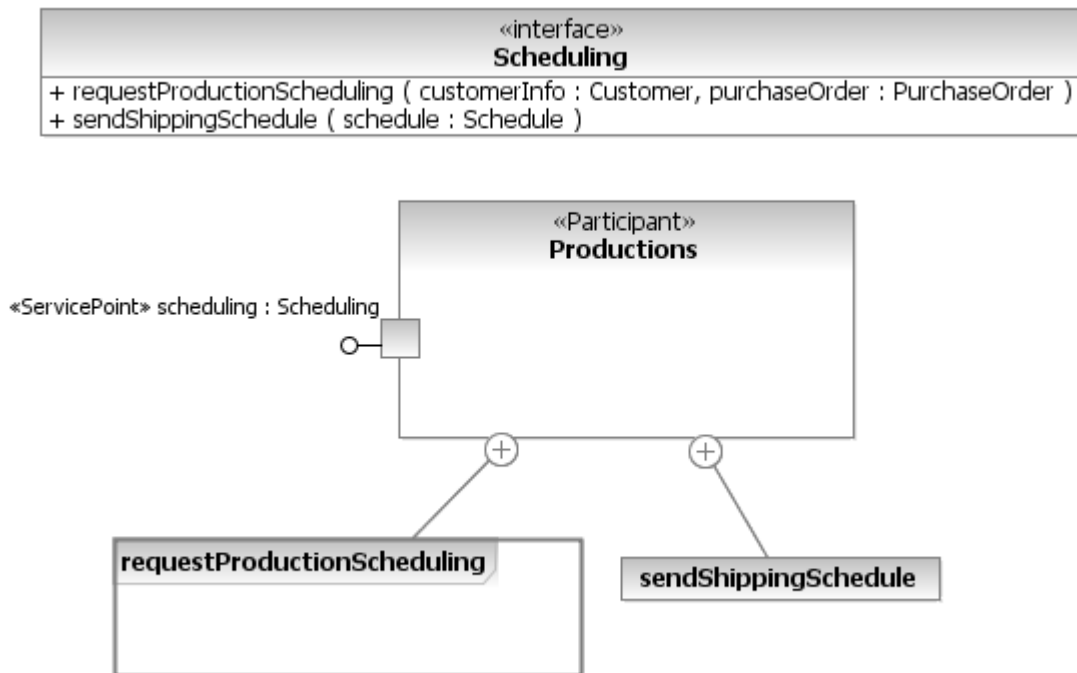


Figure 1: Services and Service Participants

Service Contracts



Figure 2: Example ServiceContract structural diagram

A ServiceContract defines the terms, conditions, interfaces and choreography that interacting participants must agree to (directly or indirectly) for the service to be enacted - the full specification of a service which includes all the information, choreography and any other “terms and conditions” of the service. A ServiceContract is binding on *both* the providers and consumers of that service. The basis of the service contract is also a UML collaboration that is focused on the interactions involved in providing a service. A participant plays a role in the larger scope of a ServicesArchitecture and also plays a role as the provider or user of services specified by ServiceContracts.

Service contracts are used in architectures where the one-way and “flat” nature of the UML interface is not sufficiently expressive or when a services architecture is going to be specified.

Each role, or party involved in a ServiceContract is defined by a ServiceInterface which is the type of the role. A ServiceContract is a binding contract – binding on any participant that has a service point typed by a role in a service contract.

An important part of the ServiceContract is the choreography. The choreography is a specification of what is transmitted and when it is transmitted between parties to enact a service exchange. The choreography specifies exchanges between the parties – the data, assets and obligations that go between the parties. The choreography defines what happens between the provider and consumer participants without defining their internal processes – their internal processes do have to be compatible with their ServiceContracts.

A ServiceContract choreography is a UML Behavior such as may be shown on an interaction diagram or activity diagram that is owned by the ServiceContract (Figure 3). The choreography defines what must go between the contract roles as defined by their service interfaces—when, and how each party is playing their role in that service without regard for who is participating. The service contract *separates the concerns* of how all parties agree to provide or use the service from how any party implements their role in that service – or from their internal business process.

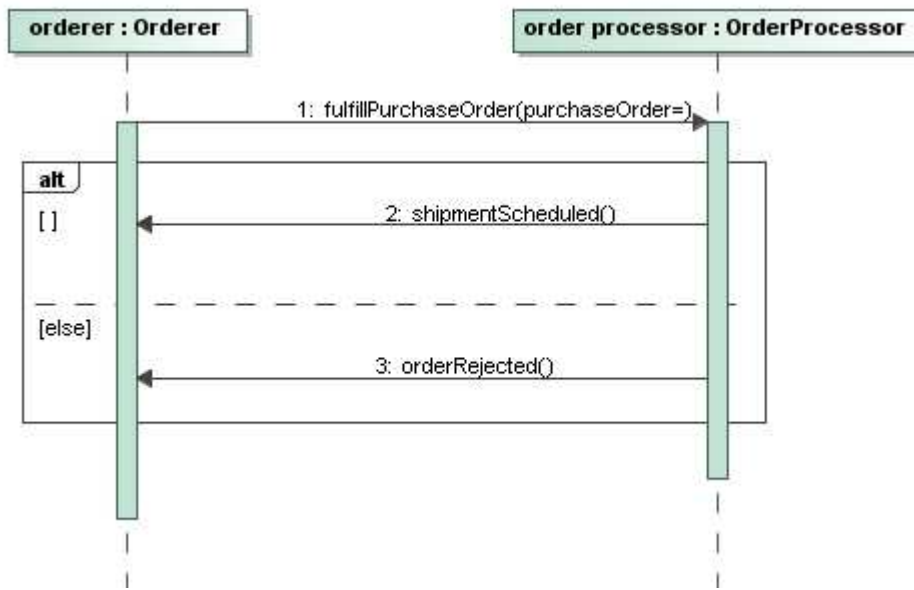


Figure 3: Example choreography

The requirements for entities playing the roles in a ServiceContract are defined by ServiceInterfaces used as the type of the role. The ServiceInterface specifies the provided and required interfaces that define all of the operations or signal receptions needed for the role it types – these will be every obligation, asset or piece of data that the entity can send or receive as part of that service contract. Providing and using corresponding UML interfaces in this way “connects the dots” between the service contract and the requirements for any participant playing a role in that service as provider or consumer. Note that some “SOA Smart” UML tools might add functionality to help “connect the dots” between service contracts, service architectures and the supporting UML classes.

It should also be noted here that it is the expectation of SoaML that services may have communications going both ways – from provider to consumer and consumer to provider and that these communications may be long-lived and asynchronous. The simpler concept of a request-response function call or invocation of an “Object Oriented” Operation is a degenerate case of a service, and can be expressed easily by just using a UML operation and a CallOperationAction. In addition, enterprise level services may be composed from simpler services. These compound services may then be delegated in whole or in part to the internal business process, technology components and participants.

Participants can engage in a variety of contracts. What connects participants to particular service contract is the use of a *role* in the context of a ServicesArchitecture. Each time a ServiceContract is used in a ServicesArchitecture; there must also be a compliant Service port on a participant – a ServicePoint or RequestPoint. This is where the participant actually offers or uses the service.

One of the important capabilities of SOA is that it can work “in the large” where independent entities are interacting across the Internet to internal departments and processes. This suggests that there is a way to decompose a ServicesArchitecture and visualize how services can be implemented by using still other services. A participant can be further described by its internal architecture, the participant architecture. Such an architecture can also use internal or external services, participants, business processes and other forms of implementation. Our concern here is to show how the internal structure of a service participant is described using other services. This is done by defining a *participant* ServicesArchitecture for participants in a more granular (larger scale) services architecture as is shown in Figure 4 and Figure 5.

The specification of a SOA is presented as a UML model and those models are generally considered to be static, however any of SoaML constructs could just as well be constructed dynamically in response to changing conditions. The semantics of SoaML are independent of the design-time, deploy-time, or run-time decision. For example, a new or specialized ServiceContract could be negotiated on the fly and

immediately used between the specific Participants. The ability of technology infrastructures to support such dynamic behavior is just emerging, but SoaML can support it as it evolves.

Key Concepts of the Services Architecture

One of the key benefits of SOA is the ability to enable a community or organization to work together more cohesively using services without getting overly coupled. This requires an understanding of how people, organizations and systems work together, or collaborate, for some purpose. We enable this collaboration by creating a services architecture model. The services architecture puts a set of services in context and shows how participants work together for a community or organization.

A ServicesArchitecture (or SOA) is a network of participant roles *providing* and *consuming services* to fulfill a purpose. The services architecture defines the requirements for the types of participants and service realizations that fulfill those roles.

Since we want to model how these people, organizations and systems collaborate without worrying, for now, about what they are, we talk about the *roles* these *participants* play in *services architectures*. A *role* defines the basic function (or set of functions) that an entity may perform *in a particular context*; in contrast, a Participant specifies the type of a party that fills the role in the context of a specific services architecture. Within a ServicesArchitecture, participant roles provide and employ any number of services. The purpose of the services architecture is to specify the SOA of some organization, community or process to provide mutual value. The participants specified in a ServicesArchitecture provide and consume services to achieve that value. The services architecture may also have a *business process* to define the tasks and orchestration of providing that value. The services architecture is a high-level view of how services work together for a purpose. The same services and participants may be used in many such architectures providing reuse.

A services architecture has components at two levels of granularity: The *community* services architecture is a “top level” view of how independent participants work together for some purpose. The services architecture of a community does not assume or require any one controlling entity or process. The services architecture of a community is modeled as collaboration stereotyped as <<ServicesArchitecture>>. A participant may also have services architecture – one that shows how parts of that participant (e.g., departments within an organization) work together to provide the services of the owning participant. The services architecture of a participant is shown as a UML structured class or component with the <<ParticipantArchitecture>> stereotype and frequently has an associated business process.

A *community* ServicesArchitecture (see Figure 4) is defined using a UML Collaboration.

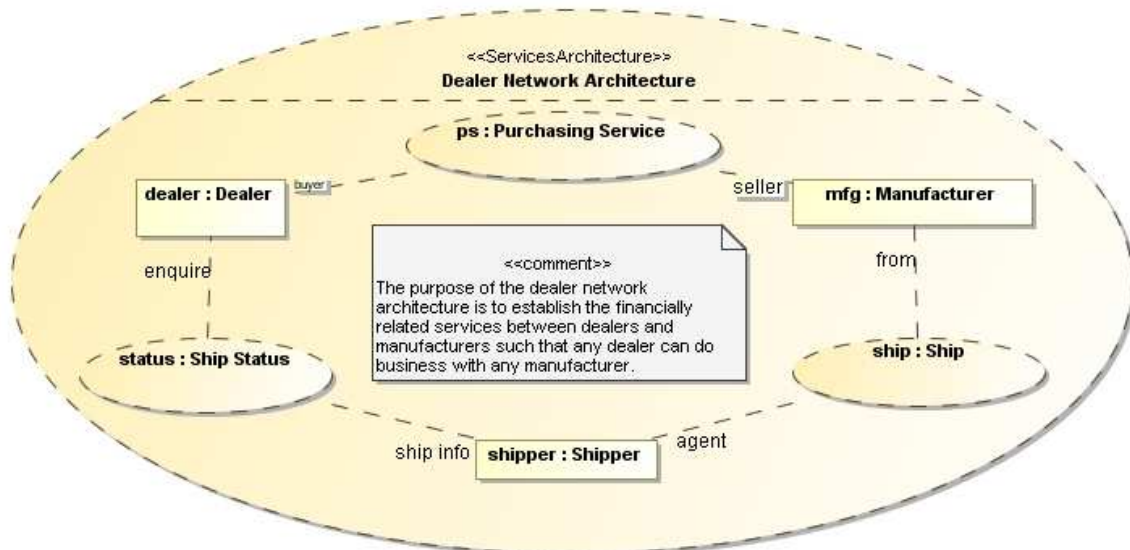


Figure 4: Example community services architecture with participant roles and services

The purpose of services architecture collaboration is to illustrate how kinds of entities work together for some purpose. Collaborations are based on the concepts of roles to define how entities are involved in that collaboration (how and why they collaborate) without depending on what kind of entity is involved (e.g. a person, organization or system). As such, we can say that an entity “plays a role” in a collaboration. The services architecture serves to define the requirements of each of the participants. . The participant roles are filled by *participants* with *service points* required of the entities that fill these roles and are then bound by the services architectures in which they participate.

A services architecture diagram can also specify the architecture for a particular Participant. Within a participant, where there is a concept of “management” exists, a participant architecture illustrates how sub-participants and external collaborators work together and would often be accompanied by a business process. A Services Architecture (both community and participant) may be composed from other services architectures and service contracts. As shown in Figure 5, Participants are classifiers defined both by the roles they play in services architectures (the participant role) and the “contract” requirements of entities playing those roles. Each participant type may “play a role” in any number of services architecture, as well as fulfill the requirements of each. Requirements are satisfied by the participant having service points that have a type compatible with the services they must provide and consume.

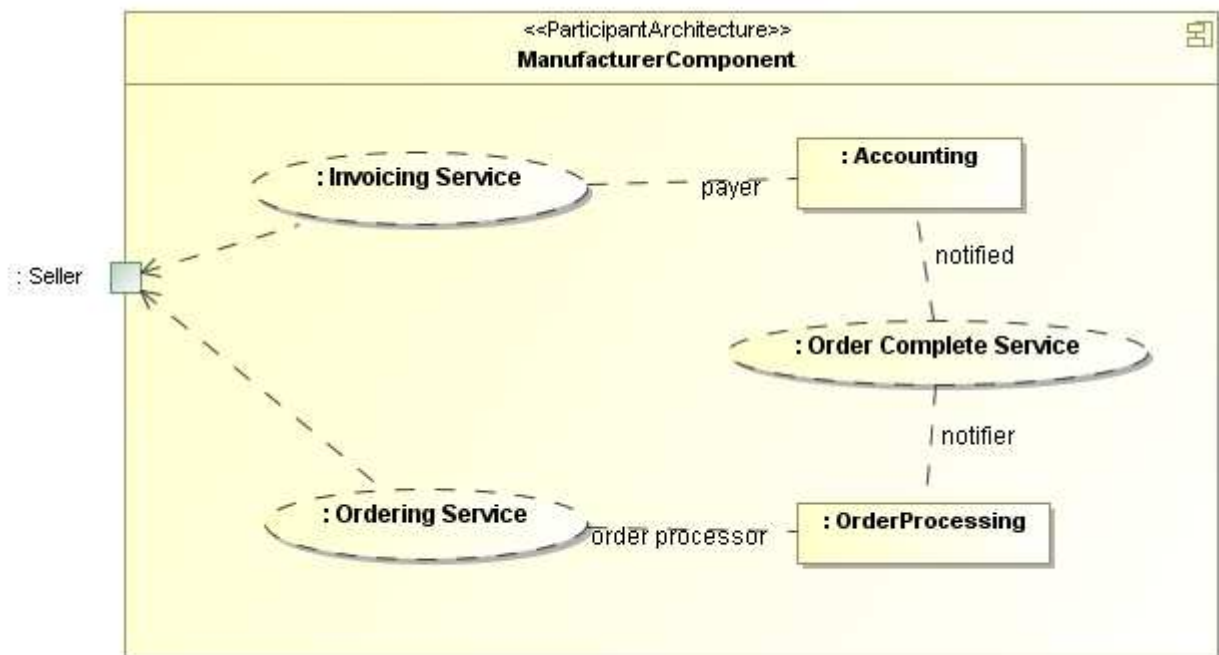


Figure 5: Example Services architecture for a participant

Figure 5 illustrates the participant services architecture for a “Manufacturer”. It indicates that this architecture consists of a number of other participants interacting through service contracts. The Manufacture participant services architecture would include a business process that specifies how these participants interact in order to provide a purchasing service. Note that other manufacturers may have different internal participant architectures but will have the same responsibilities and interfaces in the services architecture. Separating the concerns of the “inside” Vs. the “outside” is central to SOA and good architecture in general.

Service Interfaces

Like a UML interface, a ServiceInterface can be the type of a service point. The service interface has the additional feature that it can specify a bi-directional service – where both the provider and consumer have responsibilities to send and receive messages and events. The service interface is defined from the

perspective of the service provider using three primary sections: the provided and required Interfaces, the ServiceInterface class and the protocol Behavior.

The provided and required Interfaces are standard UML interfaces that are realized or used by the ServiceInterface. The interfaces that are realized specify the provided capabilities, the messages that will be received by the provider (and correspondingly sent by the consumer). The interfaces that are used by the ServiceInterface define the required capabilities, the messages or events that will be received by the consumer (and correspondingly sent by the provider). Typically only one interface will be provided or required, but not always.

The enclosed parts of the ServiceInterface represent the roles that will be played by the connected participants involved with the service. The role that is typed by the realized interface will be played by the service provider; the role that is typed by the used interface will be played by the consumer.

The Behavior specifies the valid interactions between the provider and consumer – the communication protocol of the interaction, without specifying how either party implements their role. Any UML behavior specification can be used, but interaction and activity diagrams are the most common.

The contract of interaction required and provided by a ServicePoint or RequestPoint (see below) are defined by their type which is a ServiceInterface, or in simple cases, a UML2 Interface. A ServiceInterface specifies the following information:

- The name of the service indicating what it does or is about
- The provided service interactions through realized interfaces
- The needs of consumers in order to use the service through the used interfaces
- A detailed specification of each exchange of information, obligations or assets using an operation or reception; including its name, preconditions, post conditions, inputs and outputs and any exceptions that might be raised
- Any protocol or rules for using the service or how consumers are expected to respond and when through an owned behavior
- Rules for how the service must be implemented by providers
- Constraints that can determine if the service has successfully achieved its intended purpose
- If exposed by the provider, what capabilities are used to provide the service.

This is the information potential consumers would need in order to determine if a service meets their needs and how to use the service if it does. It also specifies the responsibilities of service providers need in order to implement the service.

Figure 6 shows a generic or archetype ServiceInterface. Interface1 is the provided interface defining the capabilities, while Interface2 is the required interface defining what consumers are expected to do in response to service requests. The ServiceInterface has two parts, part1 and part2 which represent the endpoints of connections between consumer requests and provider services. Part1 has type Interface1 indicating it represents the provider or service side of the connection. Part2 has type Interface2 indicating it represents the consumer or Request side of the connection.

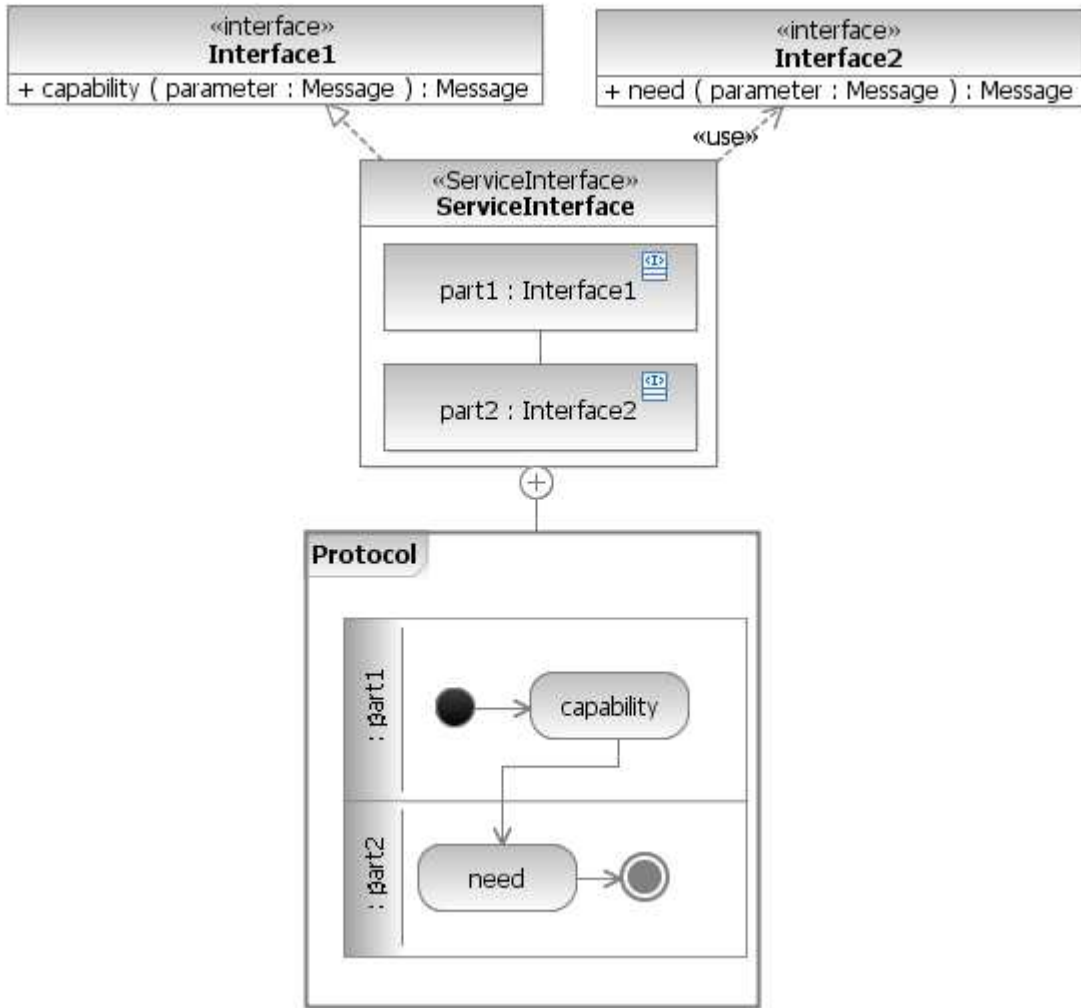
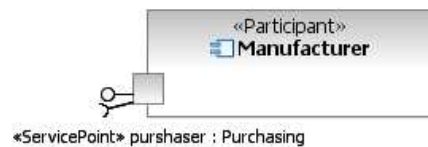


Figure 6: A generic service interface

The protocol for using the service is given in the Activity that is an owned behavior of the service interface. This activity shows the order in which the actions of the provided and required Interfaces must be called. The consumer's use of this service interface and a provider's implementation must be consistent with this protocol.

Participants and Service Points

Participants represent software components, organizations, systems or individuals that provide and use services. Participants define types of organizations, organizational roles or components by the roles they play in services architectures and the services they provide and use. For example, the figure to the right, illustrates a Manufacturer participant that offers a purchasing service. Participants provide capabilities through Service points typed by ServiceInterfaces or in simple cases, UML Interfaces that define their provided or offered capabilities.



A service uses the UML concept of a “port” and indicates the interaction point where a classifier interacts with other classifiers (see Figure 7). A port typed by a ServiceInterface, is known as a *service point*.

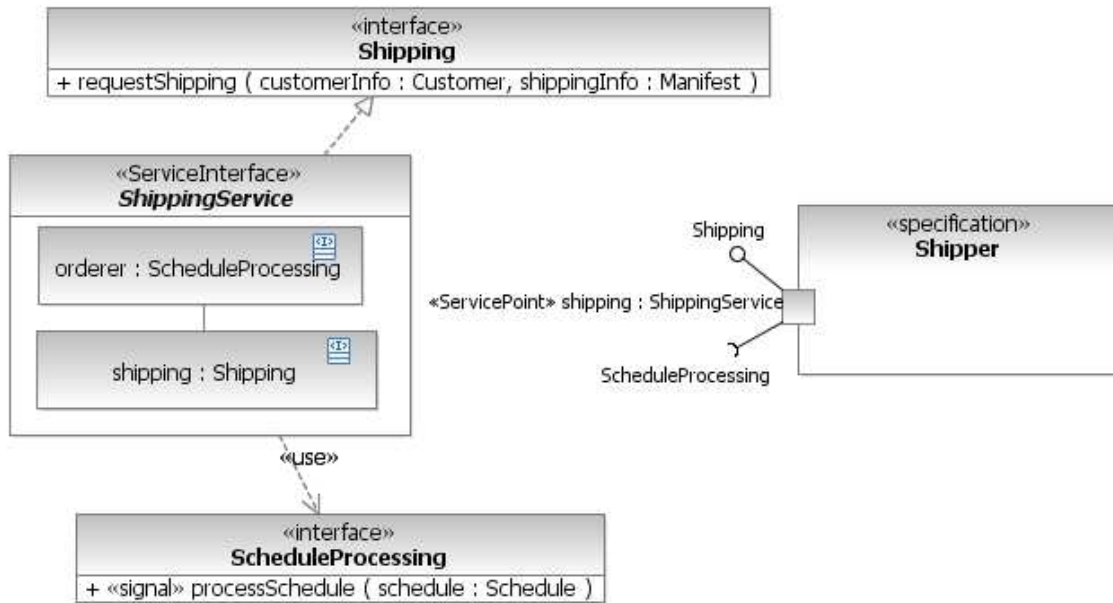


Figure 7: Example Participant with a Service Port

A *service point* is the point of interaction on a Participant where a service is actually provided or consumed. On a service provider this can be thought of as the “offer” of the service (based on the service interface). In other words, the *service point* is the point of interaction for engaging participants in a service via its service interfaces.

The Service Request

Just as we want to define the services provided by a participant using a service point, we want to define what services a participant needs or consumes. A Participant expresses their needs by making a request for services from some other Participant. A request is defined using a port stereotyped as a `<<RequestPoint>>` as shown in Figure 8.



Figure 8: A Participant with Services and Requests

The type of a Request point is also a ServiceInterface, or UML Interface, as it is with a Service port. The Request point is the conjugate of a Service point in that it defines the use of a service rather than its provision. As we will see below, this will allow us to connect service providers and consumers in a Participant.

The OrderProcessor participant example, above, shows that it provides the “purchasing” service using the “Purchasing” ServiceInterface and Requests a “shipping” service using the “ShippingService” ServiceInterface. Note that this request is the conjugate of the service that is offered by a Shipper, as shown in Figure 7.

By using service and request points, SoaML can define how both the service capabilities and needs of participants are accessed at the business or technical level.

Capability

Service architectures and service contracts provide a formal way of identifying the roles played by parties or Participants, their responsibilities, and how they are intended to interact in order to meet some objective using services. This is very useful in a “needs” or assembly context. However, when re-architecting existing applications for services or building services from scratch, the Participants may not yet known. In these situations it is also useful to express a services architecture in terms of the logical capabilities of the services in a “Participant agnostic” way. Even though service consumers should not be concerned with how a service is implemented, it is important to be able to specify the behavior of a service or capability that will realize or implement a ServiceInterface. This is done within SoaML using Capabilities.

Capabilities represent an abstraction of the ability to affect change.

They identify or specify a cohesive set of functions or resources that a service provided by one or more participants might offer. Capabilities can be used by themselves or in conjunction with Participants to represent general functionality or abilities that a Participant must have. Figure 11 shows a network of Capabilities that might be required to process an order for goods from a manufacturer. Such networks of capabilities are used to identify needed services, and to organize them into catalogues in order to communicate the needs and capabilities of a service area, whether that be business or technology focused, prior to allocating those services to particular Participants. For example, service capabilities could be organized into UML Packages to describe capabilities in some business competency or functional area. Capabilities can have usage dependencies with other Capabilities to show how these capabilities are related. Capabilities can also be organized into architectural layers to support separation of concern within the resulting service architecture.

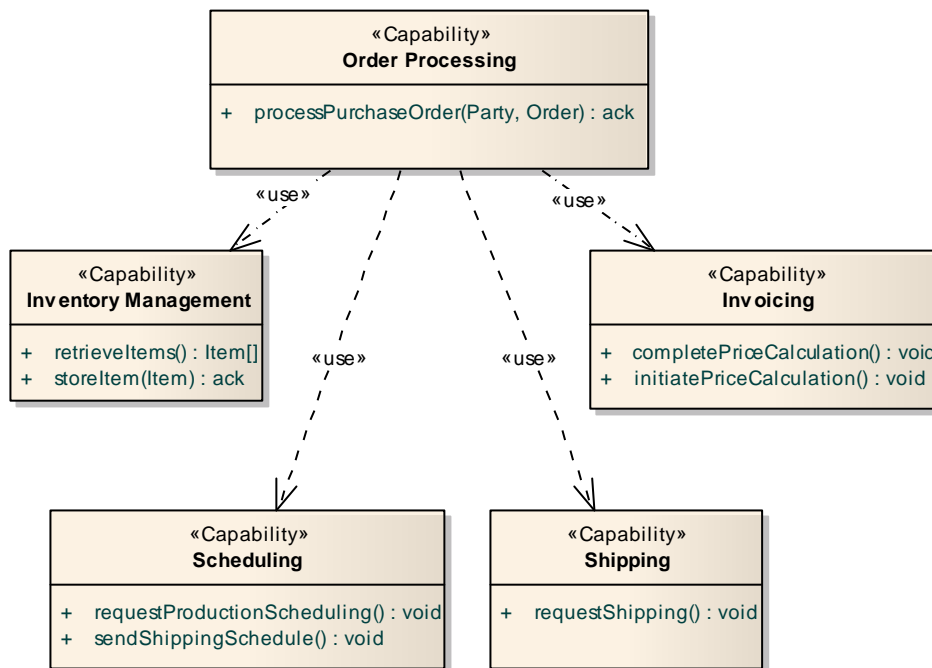


Figure 9: Service Capabilities needed for processing purchase orders

In addition to specifying abilities of Participants, one or more Capabilities can be used to specify the behavior and structure necessary to support a ServiceInterface. Figure 12 shows the Shipping Capability realizing the Shipping ServiceInterface. Thus, Capability allows for the specification of a service without regard for how that service might be implemented and subsequently offered to consumers by a Participant. It allows architects to analyze how services are related and how they might be combined to form some larger capability prior to allocation to a particular Participant.

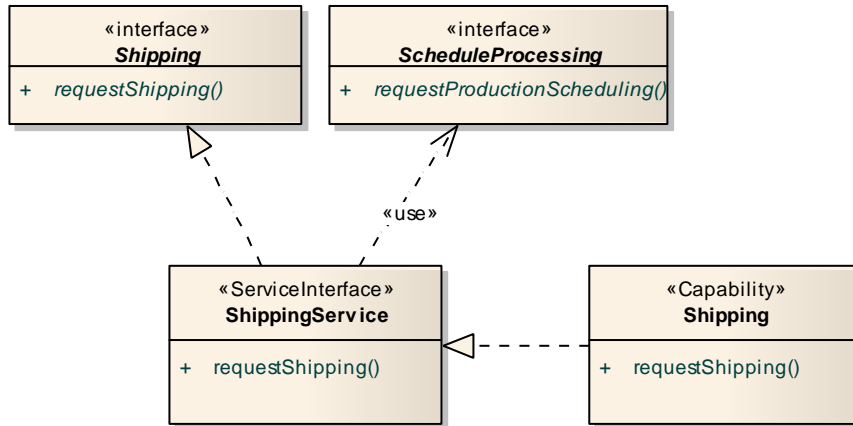


Figure 10 - ServiceInterface realized by a Capability

Capabilities can, in turn be realized by a Participant. When that Capability itself realizes a ServiceInterface, that ServiceInterface will normally be the type of a ServicePoint on the Participant as

shown in Figure 13.

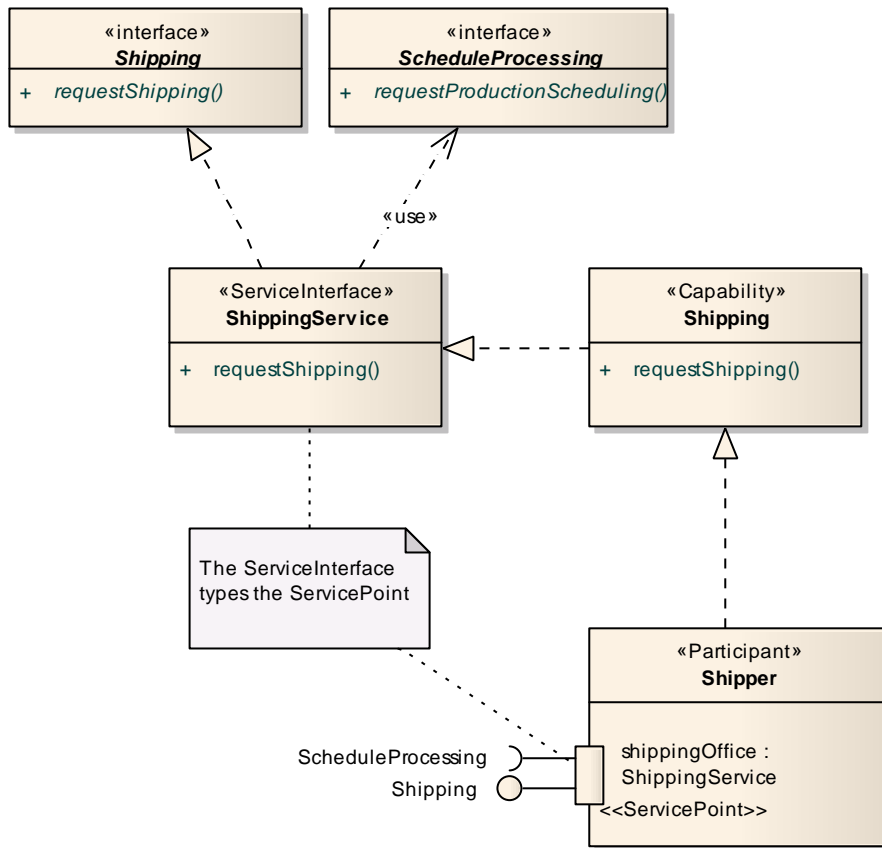


Figure 11 - The Shipper Participant realizes the Shipping Capability

Capabilities may also be used to specify the parts of Participants. Figure 14 shows the Productions Participant with two parts typed by Capabilities. The productionOffice ServicePoint delegates requests to the scheduler part that is typed by the Scheduling Capability. This would normally indicate that the Scheduling Capability realizes the SchedulingService ServiceInterface.

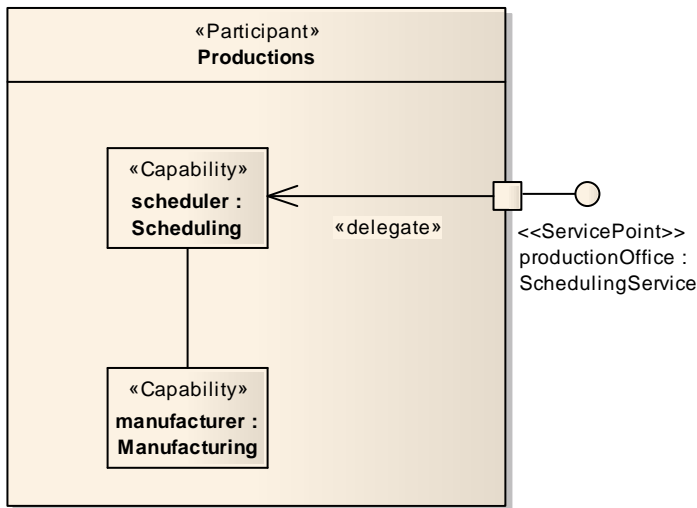


Figure 12 - Productions Participant with two parts specified by Capabilities

ServiceInterfaces may also expose Capabilities. This is done within SoaML with the Expose Dependency. While this can be used as essentially just the inverse of a Realization between a Capability and a ServiceInterface it realizes, it can also be used to represent a more general notion of “providing access” to a general capability of a Participant. Figure 15 provides an example of such a situation.

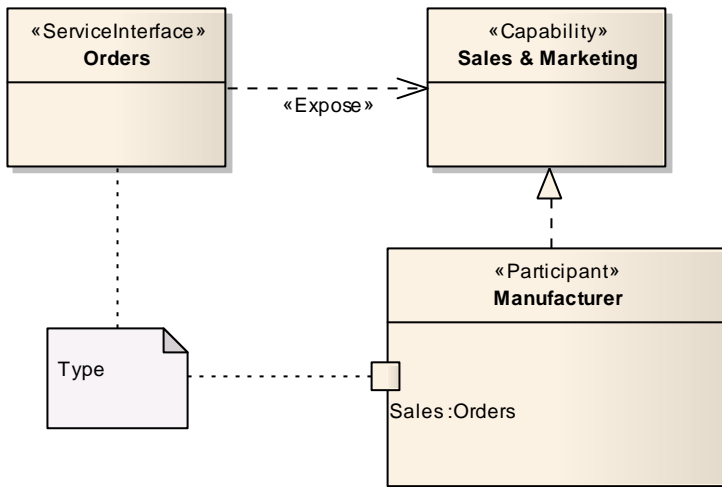


Figure 13 - The Orders ServiceInterface exposing the Sales and Marketing Capability

Each Capability may have owned behaviors that are methods of its provided Operations. These methods would be used to specify how the Capabilities might be implemented, and to identify other needed Capabilities.

Alternatively, ServiceInterfaces may simply expose Capabilities of a Participant.

The SoaML Profile of UML

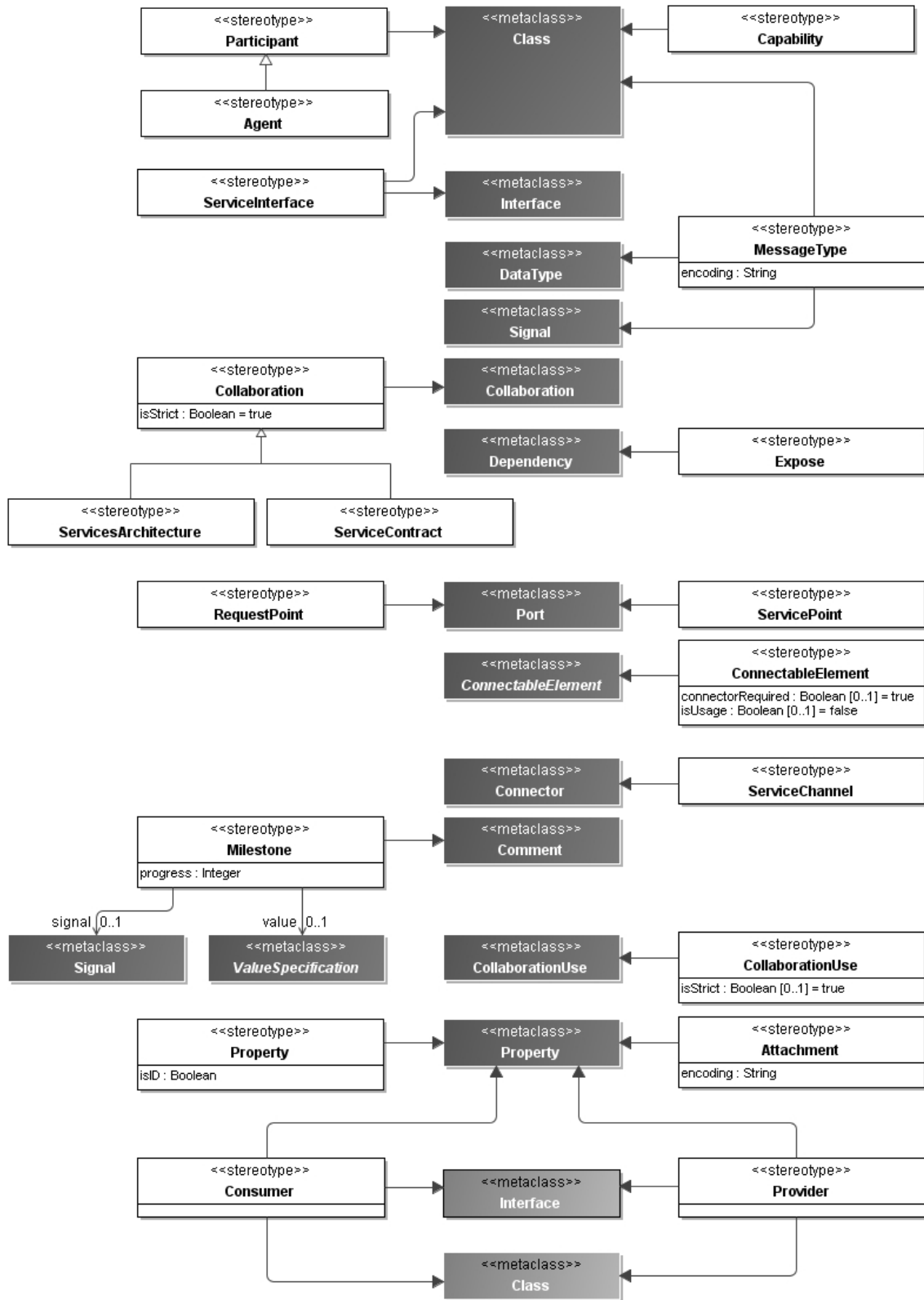


Figure 14: SoaML Profile

Stereotype Descriptions

Agent

An Agent is a classification of autonomous entities that can adapt to and interact with their environment. It describes a set of agent instances that have features, constraints, and semantics in common. Agents in SoaML are also participants, providing and using services.

Generalizes

- Participant

Description

In general, agents can be software agents, hardware agents, firmware agents, robotic agents, human agents, and so on. While software developers naturally think of IT systems as being constructed of only software agents, a combination of agent mechanisms might in fact be used from shop-floor manufacturing to warfare systems.²

These properties are mainly covered by a set of core *aspects* each focusing on different viewpoints of an agent system. Even if these aspects do not directly appear in the SoaML metamodel, we can relate them to SoaML-related concepts. Each aspect of an agent may be expressed as a services architecture.

Depending on the viewpoint of an agent system, various aspects are prominent. Even if these aspects do not directly appear in the SoaML metamodel, we can relate them to SoaML-related concepts.

- **Agent aspect** – describes single autonomous entities and the capabilities each can possess to solve tasks within an agent system. In SoaML, the stereotype Agent describes a set of agent instances that provides particular service capabilities.
- **Collaboration aspect** – describes how single autonomous entities collaborate within the multiagent systems (MAS) and how complex organizational structures can be defined. In SoaML, a ServicesArchitecture describes how aspects of agents interact for a purpose. Collaboration can involve situations such as cooperation and competition.
- **Role aspect** – covers feasible specializations and how they could be related to each role type. In SoaML, the concept of a role is especially used in the context of service contracts. Like in agent systems, the role type indicates which responsibilities an actor has to take on.
- **Interaction aspect** – describes how the interactions between autonomous entities or groups/organizations take place. Each interaction specification includes both the actors involved and the order which messages are exchanged between these actors in a protocol-like manner. In SoaML, contracts take the role of interaction protocols in agent systems. Like interaction protocols, a services contract takes a role centered view of the business requirements which makes it easier to bridge the gap between the process requirements and message exchange.

² For a further history and description of agents, see: <http://eprints.ecs.soton.ac.uk/825/05/html/chap3.htm>, http://en.wikipedia.org/wiki/Software_agent,

<http://www.sce.carleton.ca/netmanage/docs/AgentsOverview/ao.html>.

- **Behavioral aspect** – describes how plans are composed by complex control structures and simple atomic tasks such as sending a message and specifying information flows between those constructs. In SoaML, a `ServiceInterface` is a `BehavioredClassifier` and can thus contain owned `Behaviors` that can be represented by UML2 Behaviours in the form of an `Interaction`, `Activity`, `StateMachine`, `ProtocolStateMachine`, or `OpaqueBehavior`.
- **Organization/Group aspect** – Agents can form social units called groups. A group can be formed to take advantage of the synergies of its members, resulting in an entity that enables products and processes that are not possible from any single individual.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

The property `isActive` must always be true.

Semantics

The purpose of an Agent is to specify a classification of autonomous entities (agent instances) that can adapt to and interact with their environment, and to specify the features, constraints, and semantics that characterize those agent instances.

Agents deployed for IT systems generally should have the following three important properties:

- **Autonomous** - is capable acting without direct external intervention. Agents have some degree of control over their internal state and can act based on their own experiences. They can also possess their own set of internal responsibilities and processing that enable them to act without any external choreography. As such, they can act in reactive and proactive ways. When an agent acts on behalf of (or as a proxy for) some person or thing, its autonomy is expected to embody the goals and policies of the entity that it represents. In UML terms, agents can have classifier behavior that governs the lifecycle of the agent.
- **Interactive** - communicates with the environment and other agents. Agents are interactive entities because they are capable of exchanging rich forms of messages with other entities in their environment. These messages can support requests for services and other kinds of resources, as well as event detection and notification. They can be synchronous or asynchronous in nature. The interaction can also be conversational in nature, such as negotiating contracts, marketplace-style bidding, or simply making a query. In the Woodriddle-Jennings definition of agency, this property is referred to as *social ability*.
- **Adaptive** - capable of responding to other agents and/or its environment. Agents can react to messages and events and then respond in a timely and appropriate manner. Agents can be designed to make difficult decisions and even modify their behavior based on their experiences. They can learn and evolve. In the Woodriddle-Jennings definition of agency, this property is referred to as *reactivity* and *proactivity*.

Agent extends `Participant` with the ability to be active, participating components of a system. They are specialized because they have their own thread of control or lifecycle. Another way to think of agents is that they are “active participants” in an SOA system. Participants are Components whose capabilities and needs are static. In contrast, Agents are Participants whose needs and capabilities may change over time.

In SoaML, Agent is a `Participant` (a subclass of `Component`). A `Participant` represents some concrete `Component` that provides and/or consumes services and is considered an active class (`isActive=true`). However, SoaML restricts the `Participant`'s classifier behavior to that of a constructor, not something that is

intended to be long-running, or represent an “active” lifecycle. This is typical of most Web Services implementations as reflected in WS-* and SCA.

Agents possess the capability to have services and Requests and can have internal structure and ports. They collaborate and interact with their environment. An Agent's classifierBehavior, if any, is treated as its lifecycle, or what defines its emergent or adaptive behavior.

Notation

An Agent can be designated using the Component or Class/Classifier notation including the «agent» keyword. It can also be represented by stick “agent” man icon with the name of the agent in the vicinity (usually above or below) the icon. (Figure 15) Note: the stick icon is similar to the notation for UML actors, except that it has a wizard hat shaped as an elongated “A”.



Figure 15: Agent notation.

Additions to UML2

Agent is a new stereotype in SoaML extending UML2 Component with new capabilities.

Attachment

A part of a Message that is attached to rather than contained in the message.

Extends Metaclass

Property

Description

An Attachment denotes some component of a messages which is an attachment to it (as opposed to a direct part of the message itself). In general this is not likely to be used greatly in higher level design activities, but for many processes attached data is important to differentiate from embedded message data. For example, a catalog service may return general product details as a part of the structured message but images as attachments to the message; this also allows us to denote that the encoding of the images is binary (as opposed to the textual encoding of the main message). Attachments may be used to indicate part of service data that can be separately accessed, reducing the data sent between consumers and providers unless it is needed.

Attributes

- encoding: String [0..1] Denotes the platform encoding mechanism to use in generating the schema for the message; examples might be SOAP-RPC, Doc-Literal, ASN.1, etc.

Associations

No additional associations

Constraints

No additional constraints

Semantics

In an SOA supporting some business, documents may represent legally binding artifacts defining obligations between an enterprise and its partners and clients. These documents must be defined in a first class way such that they are separable from the base message and have their own identity. They can be defined using a UML2 DataType or a MessageType. But sometimes it is necessary to treat the document as a possibly large, independent document that is exchanged as part of a message, and perhaps interchanged separately. A real-world example would be all of those advertisements that fall out of your telephone statement – they are attached to the message (in the same envelope) but not part of the statement.

An Attachment extends Property to distinguish attachments owned by a MessageType from other ownedAttributes. The ownedAttributes of a MessageType must be either PrimitiveType or MessageType. The encoding of the information in a MessageType is specified by the encoding attribute. In distributed I.T. systems, it is often necessary to exchange large opaque documents in service data in order to support efficient data interchange. An Attachment allows portions of the information in a MessageType to be separated out and to have their own encoding and MIME type, and possibly interchanged on demand.

Notation

Attachments use the usual UML2 notation for DataType with the addition of an «attachment» stereotype.

Examples

Figure 16 shows an InvoiceContent Attachment to the Invoice MessageType. This attachment contains the detailed information about the Invoice.

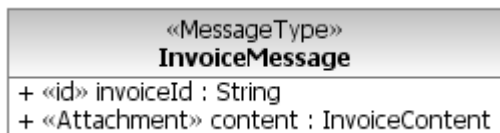


Figure 16: The InvoiceContent Attachment

Additions to UML2

Extends UML2 to distinguish message attachments from other message properties.

Capability

A Capability is the ability to act and produce an outcome that achieves a result. It can. Specify a general capability of a participant as well as the specific ability to provide a service.

Extends Metaclass

Class

Description

A Capability models the ability to act and produce an outcome that achieves a result that may provide a service specified by a ServiceContract or ServiceInterface irrespective of the Participant that might provide that service. A ServiceContract, alone, has no dependencies or expectation of how the capability is realized – thereby separating the concerns of “what” vs. “how”. The Capability may specify dependencies or

internal process to detail how that capability is provided including dependencies on other Capabilities. Capabilities are shown in context using a service dependencies diagram.

Attributes

No additional attributes.

Associations

- No additional Associations.

Constraints

No additional constraints.

Semantics

A Capability is the ability to act and produce an outcome that achieves a result. This element allows for the specification of capabilities and services without regard for the how a particular service might be implemented and subsequently offered to consumers by a Participant. It allows architects to analyze how services are related and how they might be combined to form some larger capability prior to allocation to a particular Participant.

A Capability identifies or specifies a cohesive set of functions or capabilities that a service provided by one or more participants might offer. Capabilities are used to identify needed services, and to organize them into catalogues in order to communicate the needs and capabilities of a service area, whether that be business or technology focused, prior to allocating those services to particular Participants. For example, service capabilities could be organized into UML Packages to describe capabilities in some business competency or functional area. Capabilities can have usage dependencies with other Capabilities to show how these capabilities are related. Capabilities can realize ServiceInterface and so specify how those ServiceInterfaces are supported by a Participant. Capabilities can also be organized into architectural layers to support separation of concern within the resulting service architecture.

Each capability may have owned behaviors that are methods of its provided Operations. These methods would be used to specify how the service capabilities might be implemented, and to identify other needed service capabilities. **Error! Reference source not found.** depicts the Capabilities that have been identified as needed for processing purchase orders.

Notation

A Capability is denoted using a Class or Component with the «Capability» keyword.

Examples

For examples of Capability, see **Error! Reference source not found.** and **Error! Reference source not found.**

Additions to UML2

Capability is a new stereotype used to describe service capabilities.

Consumer

The Consumer stereotype specifies an interface and/or a part as playing the role of a consumer in a provider/consumer service.

Extends Metaclass

Interface and Part (In a ServiceContract or ServiceInterface)

Description

A typical service has a “Provider” and “Consumer” to distinguish the role of the entity offering and requiring a service, respectively. The consumer typically initiates the interaction to request a service and the provider typically responds by either rejecting the service request or performing the service and returning results according to the service contract or service interface.

The consumer stereotype identifies an interface and a role in a ServiceContract or ServiceInterface as playing the “Consumer” role. A Consumer interface may then be used as the type of a RequestPoint.

Attributes

- No additional Associations.

Associations

- No additional Associations.

Constraints

A part typed by Consumer will have as its type an interface that is a Consumer.

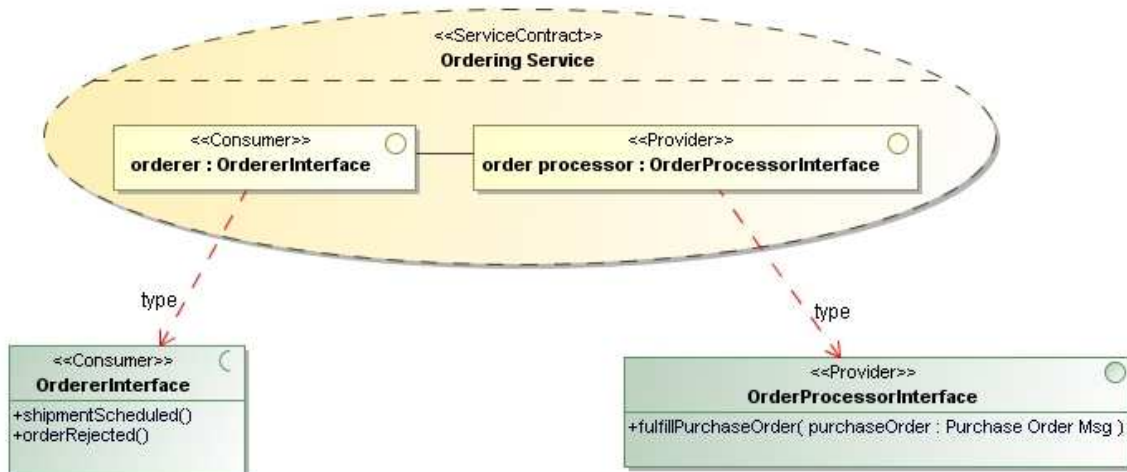
A part that is a provider must be a part of a service contract or service interface.

Semantics

A port or RequestPoint may be typed by a consumer interface to indicate that the classifier owning that port will be consuming the service specified. The consumer will initiate a message requesting that the provider provides the service. The provider will either perform the service or indicate that they are unwilling or incapable of doing so as defined by the interface. The service consumer is obligated to abide by the contract specified in the ServiceContract or ServiceInterface related to the Consumer interface.

A part typed by Consumer will have as its type an interface that is a Consumer.

Examples



CollaborationUse

CollaborationUse is extended to indicate whether the role to part bindings are strictly enforced or loose.

Extends Metaclass

CollaborationUse

Description

A CollaborationUse explicitly indicates the ability of an owning Classifier to fulfill a ServiceContract or adhere to a ServicesArchitecture. A Classifier may contain any number of CollaborationUses which indicate what it fulfills. The CollaborationUse has roleBindings that indicate what role each part in the owning Classifier plays. If the CollaborationUse is strict, then the parts must be compatible with the roles they are bound to, and the owning Classifier must have behaviors that are behaviorally compatible with the ownedBehavior of the CollaborationUse's Collaboration type.

Note that as a ServiceContract is binding on the ServiceInterfaces named in that contract, a CollaborationUse is not required if the types are compatible.

Attributes

- **isStrict:** Boolean Indicates whether this particular fulfillment is intended to be strict. A value of true indicates the roleBindings in the Fulfillment must be to compatible parts. A value of false indicates the modeler warrants the part is capable of playing the role even through the type may not be compatible.

Associations

- No new associations.

Constraints

No new constraints.

Semantics

A CollaborationUse is a statement about the ability of a containing Classifier to provide or use capabilities, have structure, or behave in a manner consistent with that expressed in its Collaboration type. It is an assertion about the structure and behavior of the containing classifier and the suitability of its parts to play roles for a specific purpose.

A CollaborationUse contains roleBindings that binds each of the roles of its Collaboration to a part of the containing Classifier. If the CollaborationUse has isStrict=true, then the parts must be compatible with the roles they are bound to. For parts to be compatible with a role, one of the following must be true:

The role and part have the same type,

The part has a type that specializes the type of the role,

The part has a type that realizes the type of the role, or

The part has a type that contains at least the ownedAttributes and ownedOperations of the role. In general this is a special case of item 3 where the part has an Interface type that realizes another Interface.

Semantic Variation Points

Compliance between types named as roles in a collaboration use is a semantic variation point and will be determined by modelers or tools.

Notation

No new notation.

Examples

The examples in section ServiceContracts describe a ServiceContract and a ServicesArchitecture. A ServiceContract is a contract describing the requirements for a specific service. A ServicesArchitecture is a Contract describing the requirements for the choreography of a collection of services or Participants.

Figure 17 shows a ShippingService ServiceInterface that fulfills the ShippingContract collaboration. The ShippingService contains a CollaborationUse that binds the parts representing the consumers and providers of the ServiceInterface to the roles they play in the ServiceContract Collaboration. The shipping part is bound to the shipping role and the orderer part is bound to the orderer role. These parts must be compatible with the roles they play. In this case they clearly are since the parts and roles have the same type. In general these types may be different as the parts will often play roles in more than one contract, or may have capabilities beyond what the roles call for. This allows ServiceInterfaces to be defined that account for anticipated variability in order to be more reusable. It also allows ServiceInterfaces to evolve to support more capabilities while fulfilling the same ServiceContracts.

The ShippingService ServiceInterface does not have to have exactly the same behavior as the ServiceContract collaboration it is fulfilling, the behaviors only need to be compatible.

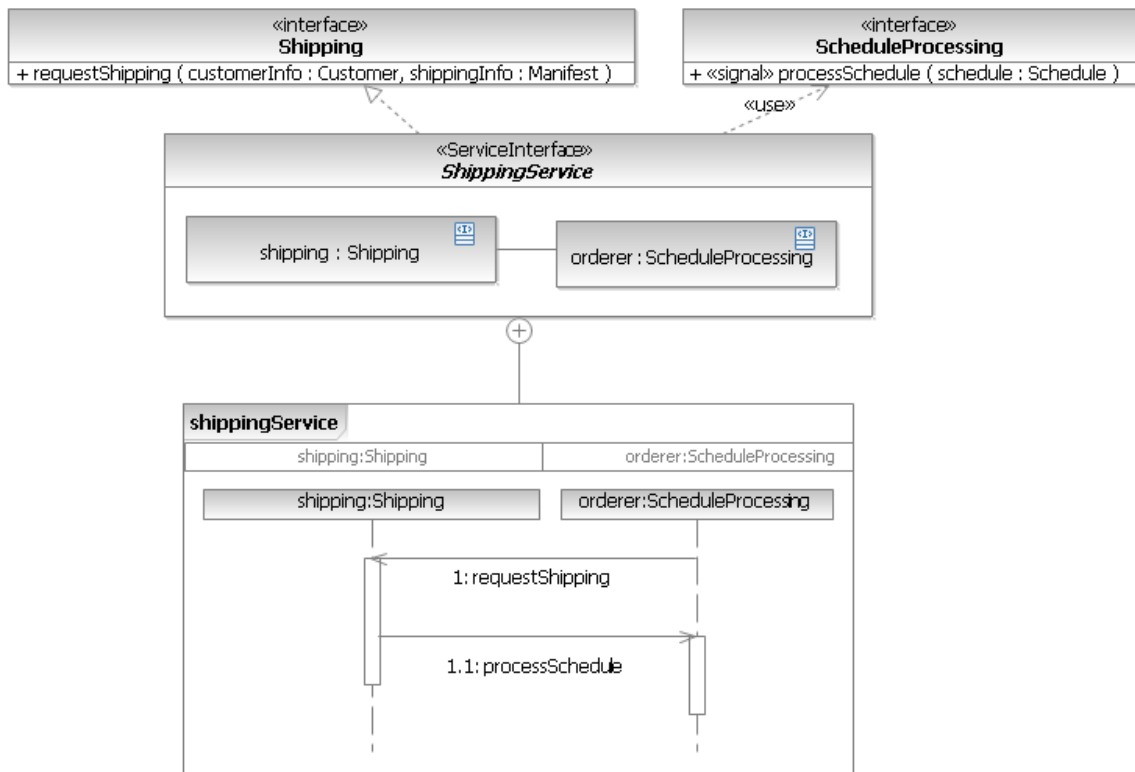


Figure 17: Fulfilling the ShippingContract ServiceContract

Figure 18 shows a Participant that assembles and connects a number of other Participants in order to fulfill the Process Purchase Order ServicesArchitecture. In this case, the roles in the ServicesArchitecture are typed by either ServiceInterfaces or Participants and the architecture specifies the expected interaction between those Participants in order to accomplish some desired result.

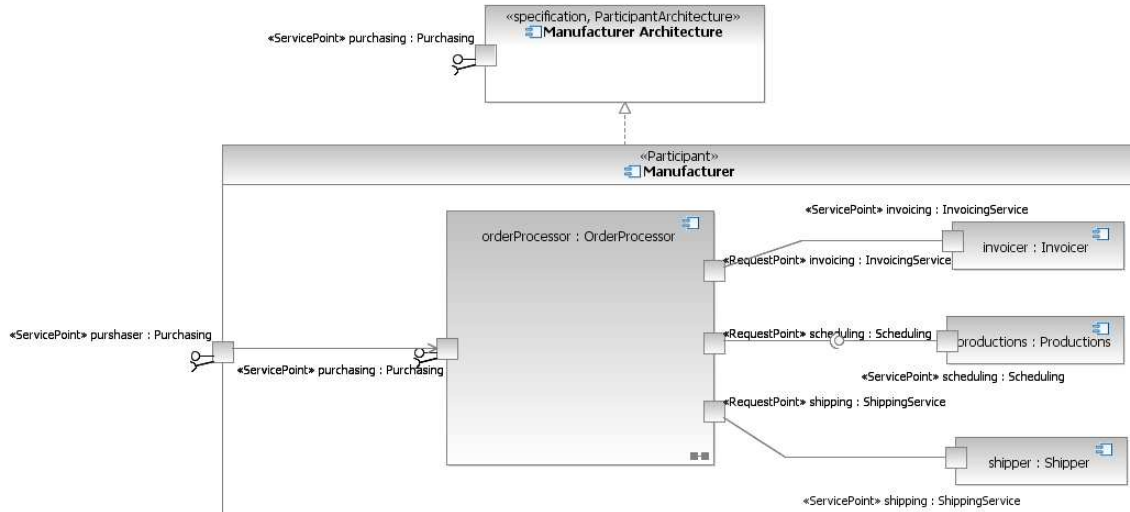


Figure 18: Fulfilling the Process Purchase Order Contract

The orderProcessor part is bound to the orderProcessor role in the ServicesArchitecture. This part is capable of playing the role because it has the same type as the role in the architecture. The invoicer part is bound to the invoicing role of the ServicesArchitecture. This part is capable of playing this role because it provides a Service whose ServiceInterface is the same as the role type in the ServicesArchitecture. The scheduling and shipping roles are similar.

Additions to UML2

CollaborationUse extends UML2 CollaborationUse to include the isStrict property.

Port

Extends UML Port with a means to indicate whether a Connection is required on this ConnectableElement or not

Extends Metaclass

Port

Description

ConnectableElement is extended with a connectorRequired property to indicate whether a connector is required on this connectable element, or the containing classifier may be able to function without anything connected

Attributes

- connectorRequired: Boolean [0..1] = true Indicates whether a connector is required on this ConnectableElement or not. The default value is true.

Associations

- No additional Associations.

Constraints

No additional constraints.

Semantics

Participants may provide many Services and have many Requests. A Participant may be able to function without all of its Services being used, and it may be able to function, perhaps with reduced qualities of service, without a services connected to all of its Requests. The property `connectorRequired` set to true on a Port indicates the Port must be connected to at least one Connector. This is used to indicate a Service point that must be used, or a Request point that must be satisfied. A Port with `connectorRequired` set to false indicates that no connection is required; the containing Component can function without interacting with another Component through that Port.

More generally, when `connectorRequired` is set to true, then all instances of this `ConnectableElement` must have a Connector or `ServiceChannel` connected. This is the default situation, and is the same as UML. If `connectorRequired` is set to false, then this is an indication that the containing classifier is able to function, perhaps with different qualities of service, or using a different implement, without any Connector connected to the part.

Notation

No additional Notation.

Additions to UML2

Adds a property to indicate whether a Connector is required on a `ConnectableElement` or not.

MessageType

The specification of information exchanged between service consumers and providers.

Extends Metaclass

`DataType`

Class

Description

A `MessageType` is a kind of value object that represents information exchanged between participant requests and services. This information consists of data passed into, and/or returned from the invocation of an operation or event signal defined in a service interface. A `MessageType` is in the domain or service-specific content and does not include header or other implementation or protocol-specific information.

`MessageTypes` are used to aggregate inputs, outputs and exceptions to service operations as in WSDL. `MessageTypes` represent “pure data” that may be communicated between parties – it is then up to the parties, based on the SOA specification, to interpret this data and act accordingly. As “pure data” message types may not have dependencies on the environment, location or information system of either party – this restriction rules out many common implementation techniques such as “memory pointers”, which may be found inside of an application. Good design practices suggest that the content and structure of messages provide for rich interaction of the parties without unnecessarily coupling or restricting their behavior or internal concerns.

The terms Data Transfer Object (DTO), Service Data Object (SDO) or value objects used in some technologies are similar in concept, though they tend to imply certain implementation techniques. A DTO represents data that can be freely exchanged between address spaces, and does not rely on specific location information to relate parts of the data. An SDO is a standard implementation of a DTO. A Value Object is a Class without identity and where equality is defined by value not reference. Also in the business world (or areas of business where EDI is commonplace) the term Document is frequently used. All these concepts can be represented by a `MessageType`.

Note: `MessageType` should generally only be applied to `DataType` since it is intended have no identity. However, it is recognized that many existing models do not clearly distinguish identity, either mixing `Class` and `DataType`, or only using `Class`. Recognizing this, SoaML allows `MessageType` to be applied to `Class` as well as `DataType`. In this case, the identity implied by the `Class` is not considered in the `MessageType`. The `Class` is treated as if it were a `DataType`.

Attributes

- `encoding: String [0..1]` Specifies the encoding of the message payload.

Associations

No additional associations

Constraints

`MessageType` cannot contain `ownedOperations`.

`MessageType` cannot contain `ownedBehaviors`.

All `ownedAttributes` must be `Public`

All `ownedAttributes` of a `MessageType` must be `PrimitiveType`, `DataType` or another `MessageType` or a reference to one of these types.

Semantics

`MessageTypes` represent service data exchanged between service consumers and providers. Service data is often a view (projections and selections) on information or domain class models representing the (often persistent) entity data used to implement service participants. `MessageType` encapsulates the inputs, outputs and exceptions of service operations into a type based on direction. A `MessageType` may contain attributes with `isID` set to true indicating the `MessageType` contains information that can be used to distinguish instances of the message payload. This information may be used to correlate long running conversations between service consumers and providers.

A service `Operation` is any `Operation` of an `Interface` provided or required by a `ServicePoint` or `RequestPoint`. Service `Operations` may use two different parameter styles, document centered (or message centered) or RPC (Remote Procedure Call) centered. Document centered parameter style uses `MessageType` for `ownedParameter` types, and the `Operation` can have at most one in, one out, and one exception parameter (an out parameter with `isException` set to true). All parameters of such an operation must be typed by a `MessageType`. For RPC style operations, a service `Operation` may have any number of in, inout and out parameters, and may have a return parameter as in UML2. In this case, the parameter types are restricted to `PrimitiveType` or `DataType`. This ensures no service `Operation` makes any assumptions about the identity or location of any of its parameters. All service `Operations` use call-by-value semantics in which the `ownedParameters` are value objects or data transfer objects.

Notation

A `MessageType` is denoted as a UML2 `DataType` with the `«messageType»` keyword.

Examples

Figure 19 shows a couple of MessageTypes that may be used to define the information exchanged between service consumers and providers. These MessageTypes may be used as types for operation parameters.

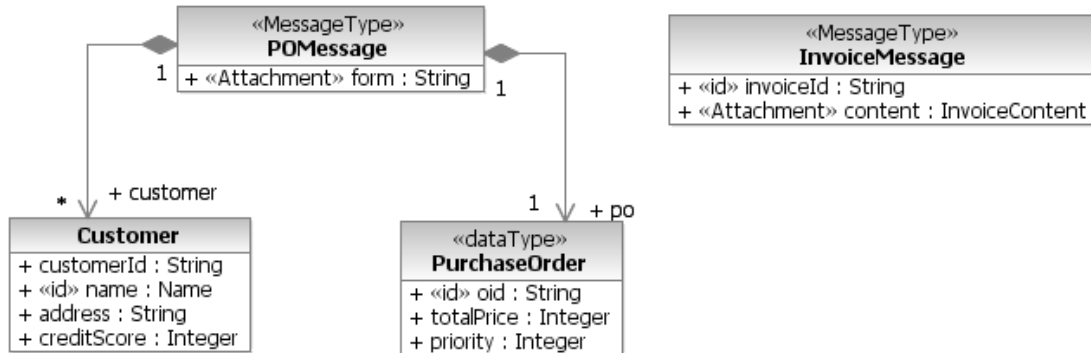


Figure 19: MessageTypes in Purchase Order Processing

MessageTypes can have associations with other message and data types as shown by the relationship between POMessage and Customer – such associations must be aggregations.

Figure 20 shows two examples of the Purchasing Interface in its processPurchaseOrder Operation. The first example uses document or message style parameters where the types of the parameters are the MessageTypes shown above. The second version uses more “Object Oriented” Remote Procedure Call (RPC) style which supports multiple inputs, outputs and a return value. The choice to use depends on modeler preference and possibly the target platform. Some platforms such as Web Services and WSDL require message style parameters, which can be created from either modeling style. It is possible to translate RPC style to message parameters in the transform, and that’s what WSDL wrapped doc-literal message style is for. But this can result in many WSDL messages containing the same information that could cause interoperability problems in the runtime platform.

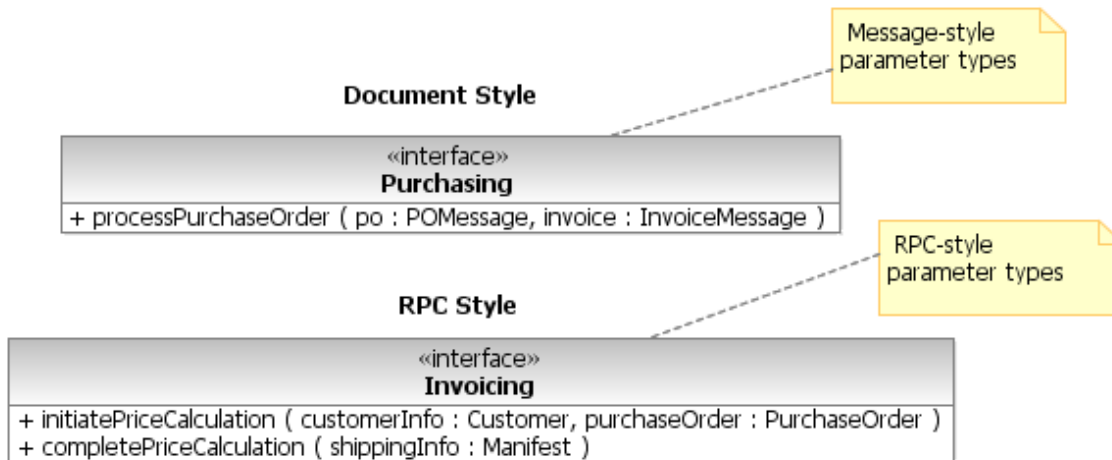


Figure 20: Document and RPC Style service operation parameters

The relationship between MessageTypes and the entity classifiers that act as their data sources is established by the semantics of the service itself. How the service parameters get their data from domain entities, and how those domain entities are updated based on changes in the parameter data is the responsibility of the service implementation.

Additions to UML2

Formalizes the notion of object used to represent pure data and message content packaging in UML2 recognizing the importance of distribution in the analysis and design of solutions.

Milestone

A Milestone is a means for depicting progress in behaviors in order to analyze liveness. Milestones are particularly useful for behaviors that are long lasting or even infinite.

Extends Metaclass

- Comment

Description

A Milestone depicts progress by defining a signal that is sent to an abstract observer. The signal contains an integer value that intuitively represents the amount of progress that has been achieved when passing a point attached to this Milestone.

Provided that a SoaML specification is available it is possible to analyze a service behavior (a Participant or a ServiceContract) to determine properties of the progress value. Such analysis results could be e.g. that the progress value can never go beyond a certain value. This could then be interpreted as a measure of the potential worth of the analyzed behaviors. In situations where alternative service behaviors are considered as in Agent negotiations, such a progress measurement could be a useful criterion for the choice.

Milestones can also be applied imperatively as specification of tracing information in a debugging or monitoring situation. The signal sent when the Milestone is encountered may contain arguments that can register any current values.

Progress values may be interpreted ordinally in the sense that a progress value of 4 is higher than a progress value of 3. A reasonable interpretation would be that the higher the possible progress value, the better. Alternatively the progress values may be interpreted nominally as they may represent distinct reachable situations. In such a case the analysis would have to consider sets of reachable values. It would typically be a reasonable interpretation that reaching a superset of values would constitute better progress possibilities.

Attributes

- progress: Integer The progress measurement.

Associations

- signal: Signal [0..1] A Signal associated with this Milestone
- value: Expression [0..1] Arguments of the signal when the Milestone is reached.

Constraints

No new constraints.

Semantics

A Milestone can be understood as a “mythical” Signal. A mythical Signal is a conceptual signal that is sent from the behavior every time a point connected to the Milestone is passed during execution. The signal is

sent to a conceptual observer outside the system that is able to record the origin of the signal, the signal itself and its progress value.

The signal is mythical in the sense that the sending of such signals may be omitted in implemented systems as they do not contribute to the functionality of the behavior. They may, however, be implemented if there is a need for run-time monitoring of the progress of the behavior.

Notation

A Milestone may be designated by a Comment with a «Milestone» keyword. The expression for the signal and signalValue is the signal name followed by the expression for the signal value in parenthesis.

Examples

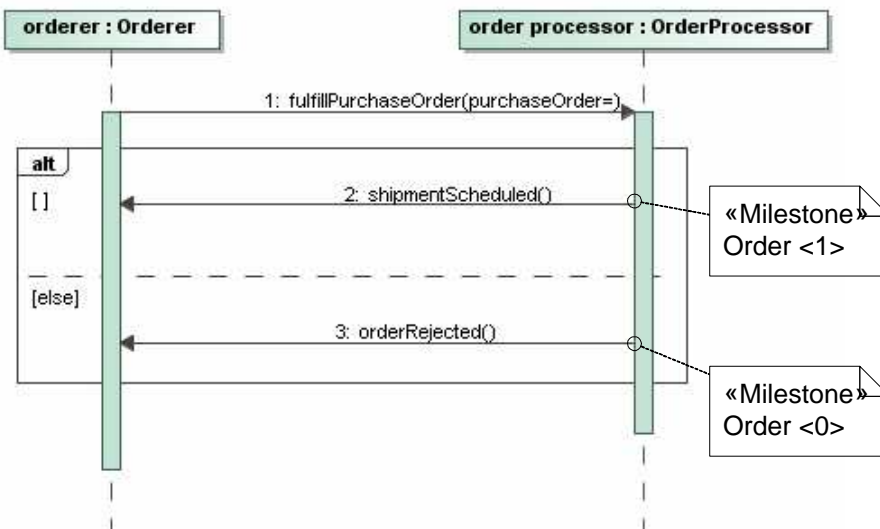


Figure 21: Milestones on Ordering Behavior

In Figure 21 we have taken the ordering behavior from Figure 3 and added Milestones to show the difference in worth between the alternatives. A seller that plays the role of order processor and only rejects order will be less worth than a seller that can be shown to provide the shipping schedule. The latter will reach the progress value 1 while the former will only be able to reach progress value 0. In both cases the signal Order will be sent to the virtual observer.

Additions to UML2

Distinguishes that this is a concept that adds nothing to the functional semantics of the behavior, and may as such be ignored by implementations.

Participant

A participant is the type of a provider and/or consumer of services. In the business domain a participant may be a person, organization or system. In the systems domain a participant may be a system, application or component.

Extends Metaclass

Class

Description

A Participant represents some (possibly concrete) party or component that provides and/or consumes services – participants may represent people, organizations or systems that provide and/or use services. A Participant is a service provider if it offers a service. A Participant is a service consumer if it uses a service – a participant may provide or consume any number of services. Service consumer and provider are roles Participants play: the role of providers in some services and consumers in others, depending on the capabilities they provide and the needs they have to carry out their capabilities. Since most consumers and providers have both services and requests, Participant is used to model both.

Participants have ServicePoint and RequestPoint ports which are the interaction points where services are offered or consumed respectively. Internally a participant may specify a behavior, a business process or a more granular service contract as a Participant Architecture.

A Participant may have CollaborationUses that indicate what ServiceContracts, ParticipantArchitectures and ServicesArchitectures it fulfills and may also define associated participant architecture. The parts of a Participant, its services and requests as well as other properties, may be bound to the roles they play in these services architectures. A concrete Participant may also realize any number of «specification» Participants (see UML2.x specification keyword and specification components).

The full scope of a SOA is realized when the relationship between participants is described using a services architecture. A services architecture shows how participants work together for a purpose, providing and using services.

Attributes

No additional attributes.

Associations

Constraints

A Participant cannot realize or use Interfaces directly, it must do so through Service and Request points.

- Note that the technology implementation of a component implementing a participant is not bound by the above rule in the case of its internal technology implementation, the connections to a participant components “container” and other implementation components may or may not use services.

Semantics

A Participant is an Agent, Person, Organization, Organizational Unit or Component that provides and/or consumes services through its Service and Request points. It represents a component that (if not a specification or abstract) can be instantiated in some execution environment or organization and connected to other participants through ServiceChannels in order to provide its services. Participants may be organizations or individuals (at the business level) or system components or agents (at the I.T. level).

A Participant implements each of its provided service operations. UML2 provides three possible ways a Participant may implement a service operation:

Method: A provided service operation may be the specification of an ownedBehavior of the Participant. The ownedBehavior is the method of the operation. When the operation is invoked by some other participant through a ServiceChannel connecting its Request to this Participant’s Service, the method is invoked and runs in order to provide the service. Any Behavior may be used as the method of an Operation including Interaction, Activity, StateMachine, ProtocolStateMachine, or OpaqueBehavior.

Event Handling: A Participant may have already running ownedBehaviors These behaviors may have forked threads of control and may contain AcceptEventAction or AcceptCallAction. An AcceptEventAction allows the Participant to respond to a triggered SignalEvent. An AcceptCallAction

allows the Participant to respond to a CallEvent. This allows Participants to control when they are willing to respond to an event or service request. Contrast with the method approach above for implementing a service operation where the consumer determines when the method will be invoked.

Delegation: A Participant may delegate a service to a service provided by one of its parts, or to a user. A part of a participant may also delegate a Request to a Request of the containing participant. This allows participants to be composed of other participants or components, and control what services and Requests are exposed. Delegation is the pattern often used for legacy wrapping in services implementations.

SoaML does not constrain how a particular participant implements its service operations. A single participant may mix delegation, method behaviors, and accept event actions to implement its services. A participant may also use different kinds of behavior to implement operations of the same service or interface provided through a service. Different concrete participants may realize or subclass the responsibilities of an abstract participant.

Semantic Variation Points

Behavioral compatibility for a ComponentRealization is a semantic variation point. In general, the actions of methods implementing Operations in a realizing Participant should be invoked in the same order as those of its realized specification Participants if any. But how this is determined based on flow analysis is not specified.

Notation

A Participant may be designated by a «participant» stereotype. Specification Participants will have both the «participant» and «specification» stereotypes.

Examples

Figure 22 shows an OrderProcessor Participant which provides the purchasing Service. This service provides the Purchasing Interface which has a single capability modeled as the processPurchaseOrder Operation. The OrderProcessor Participant also has Requests for invoicing, scheduling and shipping. Participant OrderProcessor provides a method activity, processPurchaseOrder, for its processPurchaseOrder service operation. This activity defines the implementation of the capability.

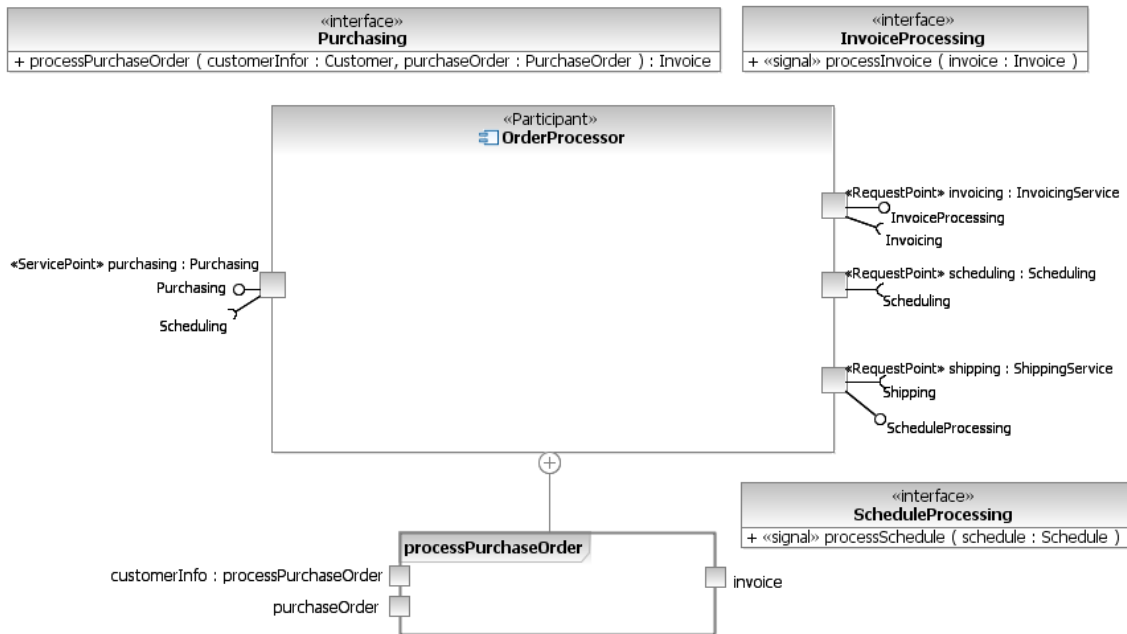


Figure 22: The OrderProcessor Participant

Figure 23 shows a Shipper specification Participant which is realized by the ShipperImpl Participant. Either may be used to type a part that could be connected to the shipping Request of the OrderProcessor Participant, but using Shipper would result in less coupling with the particular ShipperImpl implementation.

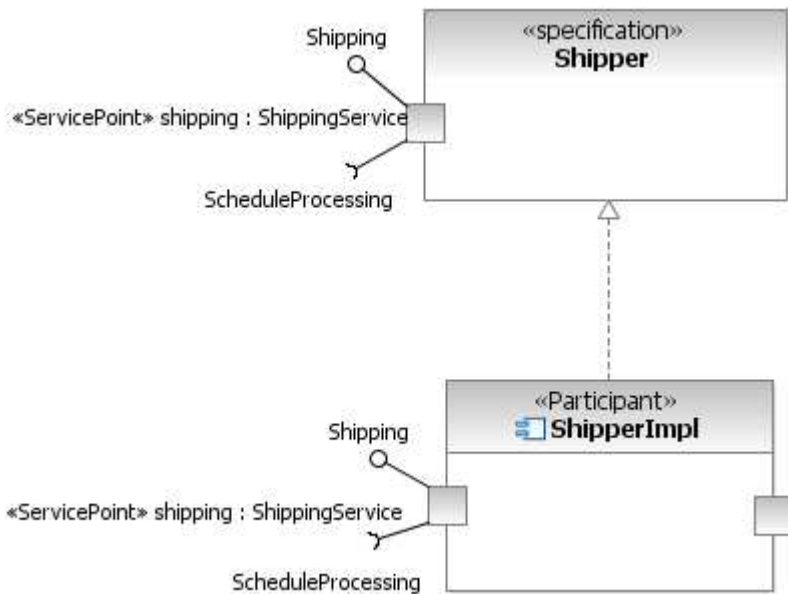


Figure 23: The Shipper specification and the ShipperImpl realization Participants

Figure 24 shows a Manufacturer Participant which is an assembly of references to other participants connected together through ServiceChannels in order to realize the Manufacturer Architecture ServicesArchitecture. The Manufacturer participant uses delegation to delegate the implementation of its purchaser service to the purchasing service of an OrderProcessor participant.

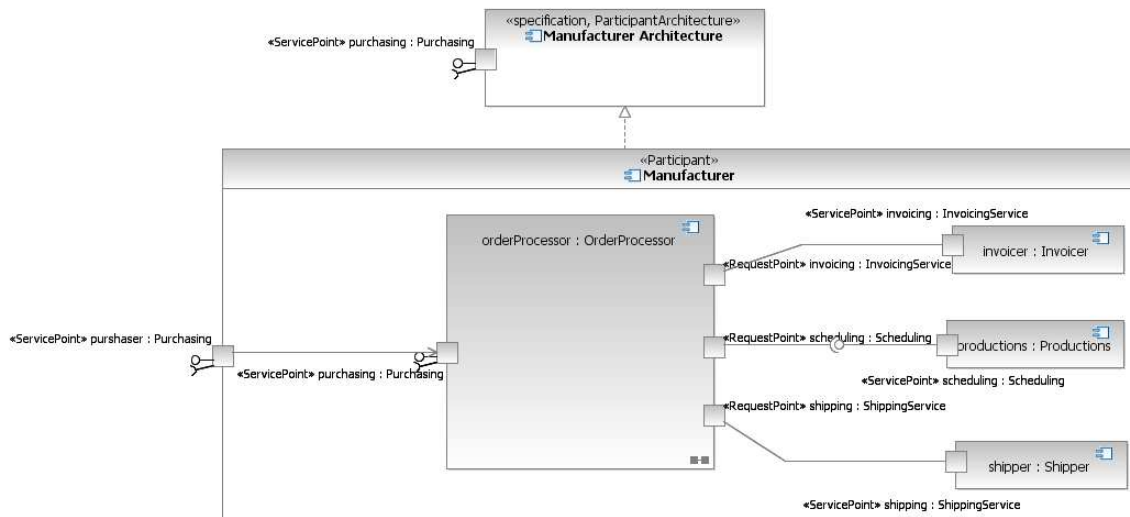


Figure 24: The Manufacturer Participant

Additions to UML2

Participant is a stereotype of UML2 Class or component with the ability to have services and requests.

Request points are introduced to make more explicit the distinction between consumed needs and provided capabilities, and to allow the same ServiceInterface to type both. This avoids the need to create additional classes to flip the realization and usage dependencies in order to create compatible types for the ports at the each end of a connector. It also avoids having to introduce the notion of conjugate types.

ParticipantArchitecture

Note: Participant Architecture is being deprecated by the SoaML Finalization Task Force in favor of using a “Services Architecture” diagram type for participant architectures. However, the examples below may be used in a services architecture diagram.

The high-level services architecture of a participant that defines how a set of internal and external participants use services to implement the responsibilities of the participant.

Extends Metaclass

Class or Component

Description

A ParticipantArchitecture describes how internal participants work together for a purpose by providing and using services expressed as service contracts. The participant architecture is a kind of services architecture (See ServicesArchitecture, below) for a particular participant. By expressing the use of services, the ParticipantArchitecture implies some degree of knowledge of the dependencies between the participants in the context of the containing participant.

A participant architecture is similar to a ServicesArchitecture detailed below and the description of such an architecture is not repeated here. The only different is that a participant architecture is based on a structured classifier rather than a collaboration and can therefore for external ports that represent interactions with external participants.

A Participant may play a role in any number of services architecture thereby representing the role a participant plays and the requirements that each role places on the participant.

Attributes

No new attributes.

Associations

- No new associations.

Constraints

The parts of a ParticipantArchitecture must be typed by a Participant or Capability. Each participant satisfying roles in a ServicesArchitecture or ParticipantArchitecture shall have a port for each role binding attached to that participant. This port shall have a type *compliant* with the type of the role used in the ServiceContract..

Semantics

Standard UML2 Collaboration semantics are augmented with the requirement that each participant used in a services architecture must have a port compliant with the ServiceContracts the participant provides or uses, which is modeled as a role binding to the use of a service contract.

Example Participant Services Architecture

Many enterprise participants are made up of smaller units that collaborate to make the enterprise work. When a participant is “decomposed” it can contain roles for other participants that also provide and use services. In this way the services architecture can start with the “supply chain” represented as B2B collaborations and drill-down to the roles within an enterprise that realize that supply chain. The services architecture of such a participant is modeled as a composite component with the «ParticipantArchitecture» stereotype.

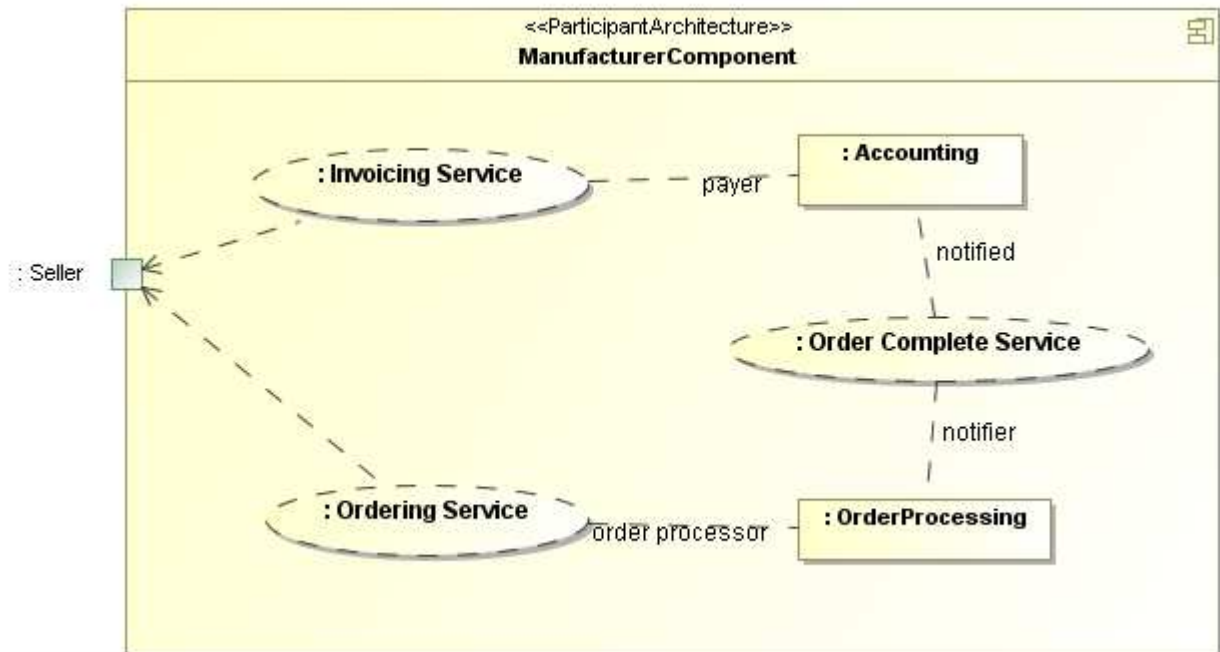


Figure 25: Participant’s services architecture

The figure, above shows a participant's services architecture. The "Manufacturer component" is composed of "Accounting" and "Order Processing". The "seller" service port on the Manufacturer component shows the external responsibility of the manufacturer which is then delegated to the accounting and order processing parts. In participant architecture there are frequently services connected between internal roles or between internal roles and external ports. The "Order CompleteService" shows a service that is internal to the Manufacturer while both the "InvoicingService" and "Ordering Service" are delegated from the Manufacturer component to the internal participants, accounting and OrderProcessing, respectively.

The business process of the manufacturer is the behavior that may be associated with the participant's services architecture. Each role in the architecture corresponds with a swim lane or pool in the business process.

Property

The Property stereotype augments the standard UML Property with the ability to be distinguished as an identifying property meaning the property can be used to distinguish instances of the containing Classifier. This is also known as a "primary key". In the context of SoaML the ID is used to distinguish the correlation identifier in a message.

Extends Metaclass

Property

Description

A property is a structural feature. It relates an instance of the class to a value or collection of values of the type of the feature. A property may be designated as an identifier property, a property that can be used to distinguish or identify instances of the containing classifier in distributed systems.

Attributes

- isID: Boolean [0..1] = false Indicates the property contributes to the identification of instances of the containing classifier.

Associations

No additional associations.

Constraints

No additional constraints

Semantics

Instances of classes in UML have unique identity. How this identity is established, and in what context is not specified. Identity is often supported by an execution environment in which new instances are constructed and provided with a system-supplied identity such as a memory address. In distributed environments, identity is much more difficult to manage in an automated, predictable, efficient way. The same issue occurs when an instance of a Classifier must be persisted as some data source such as a table in a relational database. The identity of the Classifier must be maintained in the data source and restored when the instance is reactivated in some execution environment. The instance must be able to maintain its identity regardless of the execution environment in which it is activated. This identity is often used to maintain relationships between instances, and to identify targets for operation invocations and events.

Ideally modelers would not be concerned with instance identity and persistence and distribution would be transparent at the level of abstraction supporting services modeling. Service models can certainly be created ignoring these concerns. However, persistence and distribution can have a significant impact on security, availability and performance making them concerns that often affect the design of a services architecture.

SoaML extends UML2 Property, as does MOF2, with the ability to indicate an identifying property whose values can be used to uniquely distinguish instances of the containing Classifier. This moves the responsibility of maintaining identity from the underlying execution environment where it may be difficult to handle in an efficient way to the modeler. Often important business data can be used for identification purposes such as a social security number or purchase order id. By carrying identification information in properties, it is possible to freely exchange instances into and out of persistent stores and between services in an execution environment.

A Classifier can have multiple properties with isID set to true with the set of such properties capable of identifying instance of the classifier. Compound identifiers can be created by using a Class or DataType to define a property with isID=true.

Notation

An identifying property can be denoted using the usual property notation {isID=true} or using the stereotype «id» on a property which indicates isID=true.

Examples

Figure 19 show an example of both data and message types with identifying properties.

Figure 26 shows an example of a typical Entity/Relationship/Attribute (ERA) domain model for Customer Relationship Management (CRM). These entities represent possibly persistent entities in the domain, and may be used to implement CRM services such as processing purchase orders. The id properties in these entities could for example be used to create primary and foreign keys for tables used to persist these entities as relational data sources.

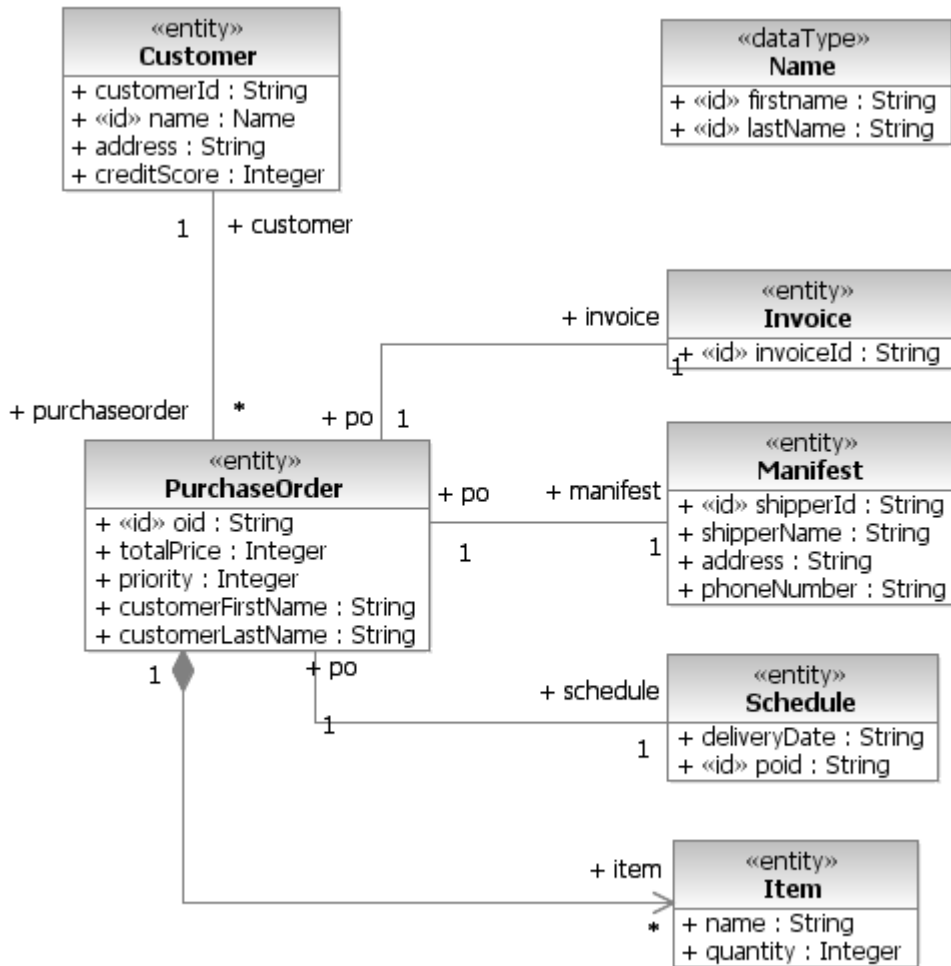


Figure 26: Example entities from the CRM domain model

Additions to UML2

Adds the isID property from MOF2 in order to facilitate identification of classifier instances in a distributed environment.

Provider

The Provider stereotype specifies an interface and/or a part as playing the role of a provider in a provider/consumer service.

Extends Metaclass

Interface and Part (In a ServiceContract or ServiceInterface)

Description

A typical service has a “Provider” and “Consumer” to distinguish the role of the entity offering and requiring a service, respectively. The consumer typically initiates the interaction to request a service and the provider typically responds by either rejecting the service request or performing the service and returning results according to the service contract or service interface.

The provider stereotype identifies an interface and a role in a ServiceContract or ServiceInterface as playing the “Provider” role. A Provider interface may then be used as the type of a ServicePoint.

Attributes

- No additional Associations.

Associations

- No additional Associations.

Constraints

A part typed by Provider will have as its type an interface that is a Provider.

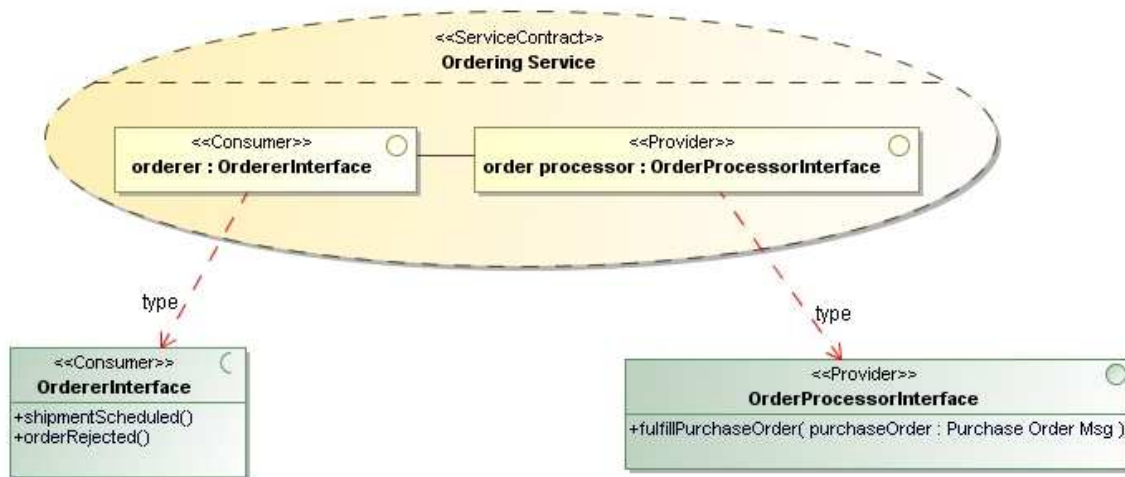
A part that is a provider must be a part of a service contract or service interface.

Semantics

A port or ServicePoint may be typed by a provider interface to indicate that the classifier owning that port will be providing the service specified for the provider. The provider will be the recipient of a message requesting that they provide the service. The provider will either perform the service or indicate that they are unwilling or incapable of doing so as defined by the interface. The service provider is obligated to abide by the contract specified in the ServiceContract or ServiceInterface related to the Provider interface.

A part typed by Provider will have as its type an interface that is a Provider.

Examples



RequestPoint

A RequestPoint models the use of a service by a participant and defines the connection point through which a Participant makes requests and uses or consumes services.

Extends Metaclass

Port

Description

A RequestPoint is interaction point through which a consumer accesses capabilities of provider services as defined by ServiceInterfaces and ServiceContracts. It includes the specification of the work required from another, and the request to perform work by another. A RequestPoint is the visible point at which provider services are connected to consumers, and through which they interact in order to produce some real world effect. A RequestPoint defines a capability that represents some functionality addressing a need, and the point of access to fulfill those needs in an execution context. The need is defined by a RequestPoint owned by a Participant and typed by a ServiceInterface or Interface. The connection point is defined by a part whose type is the ServiceInterface defining the needs.

A RequestPoint may also be viewed as some need or set of related needs required by a consumer Participant and provided by some provider Participants that has some value, or achieves some objective of the connected parties. A RequestPoint is distinguished from a simple used Operation in that it may involve a conversation between the parties as specified by some communication protocol that is necessary to meet the needs.

RequestPoint extends UML2 Port and changes how provided and required interfaces for the and are interpreted. The capabilities consumed through the RequestPoint, its required interfaces, are derived from the interfaces realized by the service's ServiceInterface. The capabilities provided by consumers in order to use the service, its provided interfaces, are derived from the interfaces used by the service's ServiceInterface. These are the opposite of the provided and required interfaces of a Port or Service and indicate the use of a Service rather than a provision of a service. Since the provided and required interfaces are reversed, a request is the *use of* the service interface – or logically the *conjugate type* of the provider.

Distinguishing requests and services allows the same ServiceInterface to be used to type both the consumer and provider ports. Any RequestPoint can connect to any ServicePoint as long as their types are compatible. Requisition and Service effectively give Ports a direction indicating whether the capabilities defined by a ServiceInterface are used or provided.

Attributes

No new attributes.

Associations

No new associations

Constraints

The type of a RequestPoint must be a ServiceInterface or an Interface

The direction property of a RequestPoint must be set to outgoing

Semantics

A RequestPoint represents an interaction point through which a consuming participant with needs interacts with a provider participant having compatible capabilities.

A RequestPoint is typed by an Interface or ServiceInterface which completely characterizes specific needs of the owning Participant. This includes required interfaces which designate the needs of the Participant through this RequestPoint, and the provided interfaces which represent what the Participant is willing and able to do in order to use the required capabilities. It also includes any protocols the consuming Participant is able to follow in the use of the capabilities through the RequestPoint.

If the type of a RequestPoint is a ServiceInterface, then the Request's provided Interfaces are the Interfaces used by the ServiceInterface while it's required Interfaces are those realized by the ServiceInterface. If the

type of a RequestPoint is a simple Interface, then the required interface is that Interface and there is no provided Interface and no protocol.

Notation

A RequestPoint may be designated by a Port with either a «RequestPoint» keyword and/or an outgoing arrow inside the RequestPoint.

Examples

Figure 27 shows an example of an OrderProcessor Participant which has a purchasing Service and three Requests: invoicing, scheduling and shipping that are required to implement this service. The implementation of the purchasing Service uses the capabilities provided through Services that will be connected to these Requests.

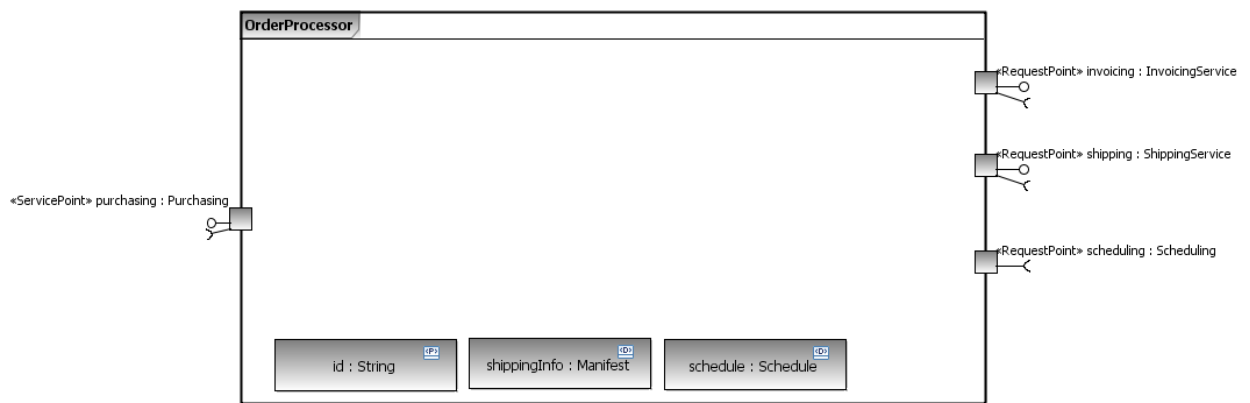


Figure 27: Requests of the OrderProcessor Participant

The invoicing Request is typed by the InvoicingService ServiceInterface. The scheduling Request is typed by the Scheduling Interface. This is an example of a simple Request that specifies simply a list of needs. It is very similar to a Reference in SCA. See section ServiceInterface for details on these service interfaces. See section Service for examples of Participants that provides services defined by these service interfaces.

Additions to UML2

A RequestPoint essentially adds directions to UML2 Ports. The purpose of a RequestPoint is to clearly distinguish between the expression of needs and the offer of capabilities. Needs and capabilities are defined by required and provided interfaces respectively. UML2 ports can do this by creating different types for ports that express the needs and capabilities by usages and realizations. However this is often inconvenient as it requires the creation of additional *conjugate types* to define compatible types for of connectable ports an each end of a connector. SoaML makes it clear whether capabilities are being provided or required through a port so that the same type may be used to define both the usage of a service and its provision.

ServicePoint

A ServicePoint is the offer of a service by one participant to others using well defined terms, conditions and interfaces. A ServicePoint defines the connection point through which a Participant offers its capabilities and provides a service to clients.

Extends Metaclass

ConnectableElement

Description

A ServicePoint is a mechanism by which a provider Participant makes available services that meet the needs of consumer requests as defined by ServiceInterfaces, Interfaces and ServiceContracts. A ServicePoint is represented by a UML Port on a Participant stereotyped as a «ServicePoint», .

By referencing a service interface a ServicePoint may include the specification of the value offered to another, and the offer to provide value to another. A service port is the visible point at which consumer requests are connected to providers, and through which they interact in order to produce some real world effect.

A ServicePoint may also be viewed as the offer of some service or set of related services provided by a provider Participant and consumed by some consumer Participants that has some value, or achieves some objective of the connected parties. A ServicePoint is distinguished from a simple Operation in that it may involve a conversation between the parties as specified by some communication protocol that is necessary to meet the common objective.

The capabilities provided through the service, its provided interfaces, are derived from the interfaces realized by the service's ServiceInterface and further detailed in the service contract. The capabilities required of consumers in order to use the service, its required interfaces, are derived from the interfaces used by the service's ServiceInterface. These are the same as the provided and required interfaces of the Port that is a generalization of ServicePoint .

Attributes

No new attributes.

Associations

No New associations.

Constraints

The type of a ServicePoint must be a ServiceInterface or an Interface

The direction property of a ServicePoint must be incoming

Semantics

A ServicePoint represents an interaction point through which a providing Participant with capabilities to provide a service interacts with a consuming participant having compatible needs. It represents a part at the end of a ServiceChannel connection and the point through which a provider satisfies a request.

A ServicePoint is typed by an Interface or ServiceInterface that, possibly together with a ServiceContract, completely characterizes specific capabilities of the producing and consuming participants' responsibilities with respect to that service. This includes provided interfaces which designate the capabilities of the Participant through this Service, and the required interfaces which represent what the Participant is requires of consumers in order to use the provided capabilities. It also includes any protocols the providing Participant requires consumers to follow in the use of the capabilities of the Service.

If the type of a ServicePoint is a ServiceInterface, then the Service's provided Interfaces are the Interfaces realized by the ServiceInterface while it's required Interfaces are those used by the ServiceInterface. If the type of a ServicePoint is a simple Interface, then the provided interface is that Interface and there is no required Interface and no protocol. If the ServiceInterface or UML interface typing a service port is defined as a role within a ServiceContract – the service port (and participant) is bound by the semantics and constraints of that service contract.

Notation

A Service may be designated by a Port with either a «ServicePoint» keyword and/or an incoming arrow inside the service port.

Examples

Figure 28 shows an invoicing service provided by an Invoicer Participant. In this example, the Invoicer Participant realizes the Invoicing UseCase that describes the high-level requirements for the service provider and its services. The invoicing Service is typed by the InvoicingService ServiceInterface which defines the interface to the service. See the section ServiceInterface for further details on this ServiceInterface.

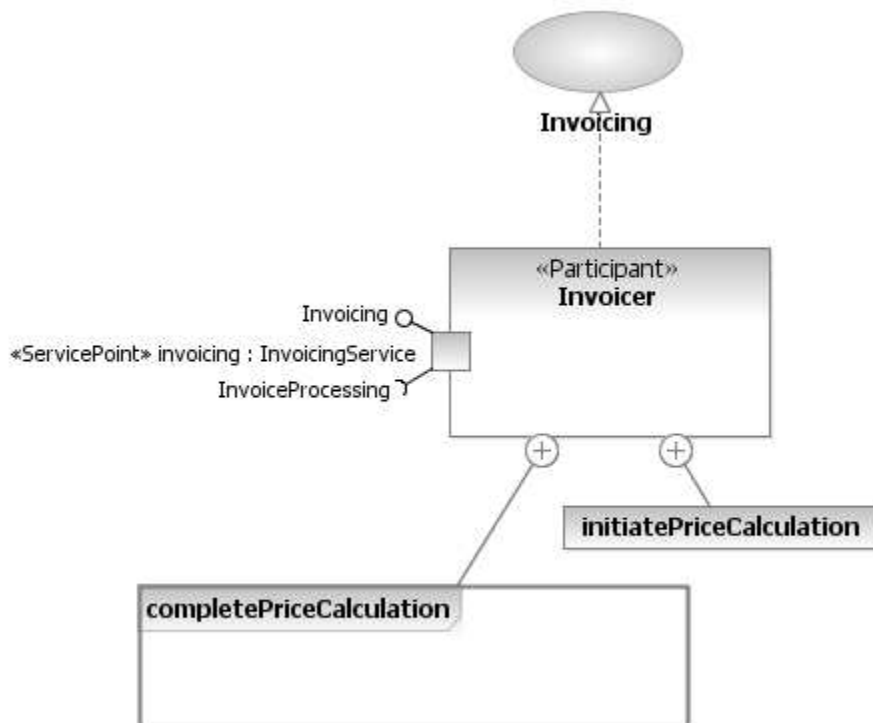


Figure 28: The invoicing Service of the Invoicer Participant

The Invoicer Participant has two ownedBehaviors, one an Activity and the other an OpaqueBehavior which are the methods for the Operations provided through the invoicing service and model the implementation of those capabilities – no stereotypes are provided as these are standard UML constructs..

Figure 29 shows an example of a scheduling Service provided by a Scheduling Participant. In this case, the type of the Service is a simple Interface indicating what capabilities are provided through the Service, and that consumers are not required to provide any capabilities and there is no protocol for using the service capabilities. SoaML allows Services type typed by a simple interface in order to support this common case.

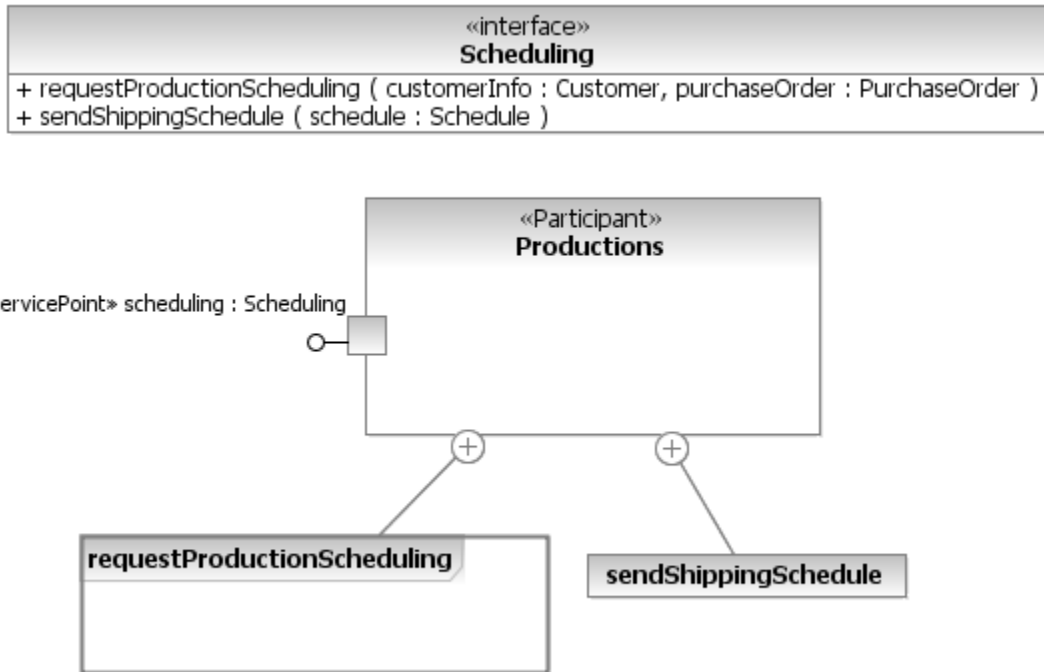


Figure 29: The scheduling Service of the Productions Participant

Productions also has ownedBehaviors which are the methods of its provided service operations.

Additions to UML2

ServiceChannel

A communication path between ServicePoints and RequestPoints within an architecture..

Extends Metaclass

Connector

Description

A ServiceChannel provides a communication path between consumer Requests and provider services.

Attributes

No new attributes.

Associations

No new associations.

Constraints

- One end of a ServiceChannel must be a RequestPoint and the other a ServicePoint in an architecture.
- The Request and Service connected by a ServiceChannel must be compatible

- The contract Behavior for a ServiceChannel must be compatible with any protocols specified for the connected requests and Services

Semantics

A ServiceChannel is used to connect Requests of consumer Participants to Services of provider Participants at the ServiceChannel ends. A ServiceChannel enables communication between the Request and service.

A RequestPoint specifies a Participant's needs. A ServicePoint specifies a Participant's services offered. The type of a RequestPoint or ServicePoint is a ServiceInterface or Interface that defines the needs and capabilities accessed by a Request through Service, and the protocols for using them. Loosely coupled systems imply that services should be designed with little or no knowledge about particular consumers. Consumers may have a very different view of what to do with a service based on what they are trying to accomplish. For example, a guitar can make a pretty effective paddle if that's all you have and you're stuck up a creek without one.

Loose coupling allows reuse in different contexts, reduces the effect of change, and is the key enabler of agile solutions through an SOA. In services models, ServiceChannels connect consumers and providers and therefore define the coupling in the system. They isolate the dependencies between consuming and providing participants to particular Request/service interaction points. However, for services to be used properly, and for Requests to be fully satisfied, Requests must be connected to compatible Services. This does not mean the Request Point and Service Point must have the same type, or that their ServiceInterfaces must be derived from some agreed upon ServiceContract as this could create additional coupling between the consumer and provider. Such coupling would for example make it more difficult for a service to evolve to meet needs of other consumers, to satisfy different contracts, or to support different versions of the same request without changing the service it is connected to.

Loosely coupled systems therefore require flexible compatibility across ServiceChannels. Compatibility can be established using UML2 specialization/generalization or realization rules. However, specialization/generalization, and to a lesser extent realization, are often impractical in environments where the classifiers are not all owned by the same organization. Both specialization and realization represent significant coupling between subclasses and realizing classifiers. If a superclass or realized class changes, then all the subclasses also automatically change while realizing classes must be examined to see if change is needed. This may be very undesirable if the subclasses are owned by another organization that is not in a position to synchronize its changes with the providers of other classifiers.

A RequestPoint is compatible with, and may be connected to a ServicePoint though a ServiceChannel if:

- The Request and Service have the same type, either an Interface or ServiceInterface
- The type of the Service is a specialization or realization of the type of the Request.
- The Request and Service have compatible needs and capabilities respectively. This means the Service must provide an Operation for every Operation used through the Request, the Request must provide an Operation for every Operation used through the Service, and the protocols for how the capabilities are compatible between the Request and Service.
- Any of the above are true for a subset of a ServiceInterface as defined by a port on that service interface.

Semantic Variation Points

Behavioral compatibility between Requests and Services is a semantic variation point.

Notation

A ServiceChannel uses the same notation as a UML2 Connector and may be shown using the «serviceChannel» keyword.

Examples

Figure 30 illustrates a Manufacturer service Participant that assembles a number of other Participants necessary to actually implement a service as a deployable runtime solution. Manufacturer provides a purchaser service that it delegates to the purchasing service of its orderProcessor part. ServiceChannels connect the Requests to the Services the OrderProcessor needs in order to execute.

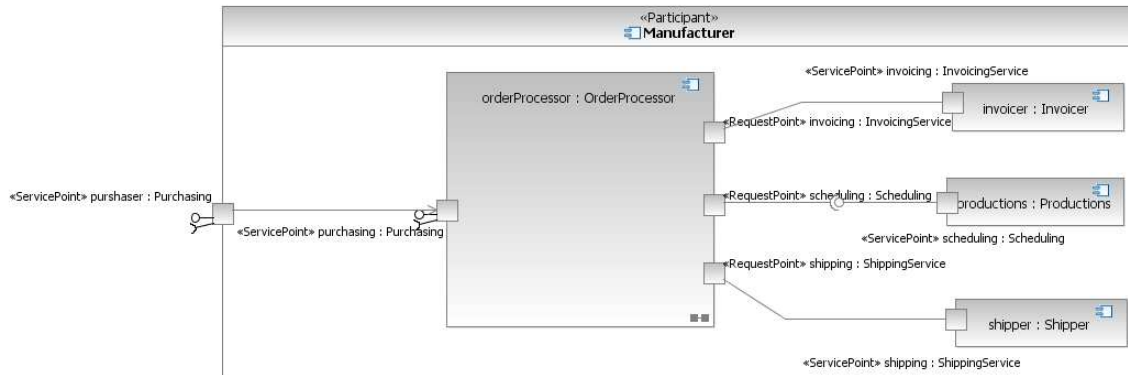


Figure 30: The Manufacturer Participant

Additions to UML2

ServiceChannel extends UML2 Connector with more specific semantics for service and request compatibility.

ServiceContract

A ServiceContract is the formalization of a binding exchange of information, goods, or obligations between parties defining a service.

Extends Metaclass

Collaboration

Description

A ServiceContract is the specification of the agreement between providers and consumers of a service as to what information, products, assets, value and obligations will flow between the providers and consumers of that service – it specifies the service without regard for realization, capabilities or implementation. A ServiceContract does not require the specification of who, how or why any party will fulfill their obligations under that ServiceContract, thus providing for the loose coupling of the SOA paradigm. In most cases a ServiceContract will specify two roles (provider and consumer) – but other service roles may be specified as well. The ServiceContract may also own a behavior that specifies the sequencing of the exchanges between the parties as well as the resulting state and delivery of the capability. The owned behavior is the *choreography* of the service and may use any of the standard UML behaviors such as an interaction, timing, state or activity diagram.

Enterprise services are frequently complex and nested (e.g., placing an order within the context of a long-term contract). A ServiceContract may use other nested ServiceContracts representing nested services as a CollaborationUse. Such a nested service is performed and completed within the context of the larger grained service that uses it. A ServiceContract using nested ServiceContracts is called a *compound service contract*.

One ServiceContract may specialize another service contract using UML generalization. A specialized contract must comply with the more general contract but may restrict the behavior and/or operations used.

A specialized contract may be used as a general contract or as a specific agreement between specific parties for their use of that service.

A `ServiceContract` is used to model an agreement between two or more parties and may constrain the expected real world effects of a service. `ServiceContracts` can cover requirements, service interactions, quality of service agreements, interface and choreography agreements, and commercial agreements.

Each service role in a service contract has a type, which must be a `ServiceInterface` or UML Interface and usually represents a provider or consumer. *The `ServiceContract` is a binding agreement on entities that implement the service type.* That is, any party that “plays a role” in a Service Contract is bound by the service agreement, exchange patterns, behavior and `MessageType` formats. Note that there are various ways to bind to or fulfill such an agreement, but compliance with the agreement is ultimately required to participate in the service. Due to the binding agreement, where the types of a service contract are used in a `ServicePoint` or `RequestPoint` no collaboration use is required.

The Service contract is at the middle of the SoaML set of SOA architecture constructs. The highest level is described as a services architectures (at the community and participant levels) – were participants are working together using services. These services are then described by a `ServiceContract`. The details of that contract, as it relates to each participant use a `ServiceInterface` which in turn uses the message data types that flow between participants. The service contract provides an explicit but high-level view of the service where the underlying details may be hidden or exposed, based on the needs of stakeholders.

A `ServiceContract` can be used in support of multiple architectural goals, including:

- As part of the Service Oriented Architecture (SOA), including services architectures, participant architectures, information models and business processes.
- Multiple views of complex systems
- A way of abstracting different aspects of services solutions
- Convey information to stakeholders and users of those services
- Highlight different subject areas of interest or concern
- Formalizing requirements and requirement fulfillment
- Without constraining the architecture for how those requirements might be realized
- Allowing for separation of concerns.
- Bridge between business process models and SOA solutions
- Separates the what from the how
- Formal link between service implementation and the contracts it fulfills with more semantics than just traceability
- Defining and using patterns of services
- Modeling the requirements for a service
- Modeling the roles the consumers and providers play, the interfaces they must provide and/or require, and behavioral constraints on the protocol for using the service.
- The foundation for formal Service Level Agreements
- Modeling the requirements for a collection of services or service participants
- Specifying what roles other service participants are expected to play and their interaction choreography in order to achieve some desired result including the implementation of a composite service
- Defining the choreography for a business process

Attributes

No new attributes.

Associations

No new associations.

Constraints

If the CollaborationUse for a ServiceInterface in a services architecture has isStrict=true (the default), then the parts must be compatible with the roles they are bound to. For parts to be compatible with a role, one of the following must be true:

- The role and part have the same type,
- The part has a type that specializes the type of the role,
- The part has a type that realizes the type of the role, or
- The part has a type that contains at least the ownedAttributes and ownedOperations of the role. In general this is a special case of item 3 where the part has an Interface type that realizes another Interface.

Semantics

Each ServiceContract role has a type that must be a ServiceInterface or UML Interface. The ServiceContract is a binding agreement on participants that implements the service type. That is, any party that “plays a role” in a Service Contract is bound by the service agreement, interfaces, exchange patterns, behavior and Message formats. Note that there are various ways to bind to or fulfill such an agreement, but compliance with the agreement is ultimately required to participate in the service as a provider or consumer.

ServiceContract shares all the semantics of UML2 Collaboration and extends those semantics by making the service contract binding on the types of the roles without a collaboration use being required. Any behavior specified as part of a ServiceContract is then a specification of how the parties that use that service must interact. By typing a port with a ServiceContract that is the type of a role in a ServiceContract the participant agrees to abide by that contract.

Where a ServiceInterface has a behavior and is also used as a type in a ServiceContract, the behavior of that ServiceInterface must comply with the service contract. However, common practice would be to specify a behavior in the service contract or service interface, not both.

Examples

In the context of services modeling, ServiceContracts may be used to model the specification for a specific service. A ServicesArchitecture or ParticipantArchitecture may then be used to model the requirements for a collection of participants that provide and consume services defined with service contracts.

When modeling the requirements for a particular service, a ServiceContract captures an agreement between the roles played by consumers and providers of the service, their capabilities and needs, and the rules for how the consumers and providers must interact. The roles in a ServiceContract are typed by Interfaces that specify Operations and events which comprise the choreographed interactions of the services.. A ServiceInterface may fulfill zero or more ServiceContracts to indicate the requirements it fulfills but they are usually one-one.

Figure 31 is an example of a ServiceContract. The orderer and order processor participate in the contract.



Figure 31: The Ordering Service Contract

The service contract diagram shows a high level “business view” of services but includes ServiceInterfaces as the types of the roles to ground the business view in the required details. While two roles are shown in the example, a ServiceContract may have any number of roles. Identification of the roles may then be augmented with a behavior. Real-world services are typically long-running, bi-directional and asynchronous. This real-world behavior shows the information and resources that are transferred between the service provider and consumer.

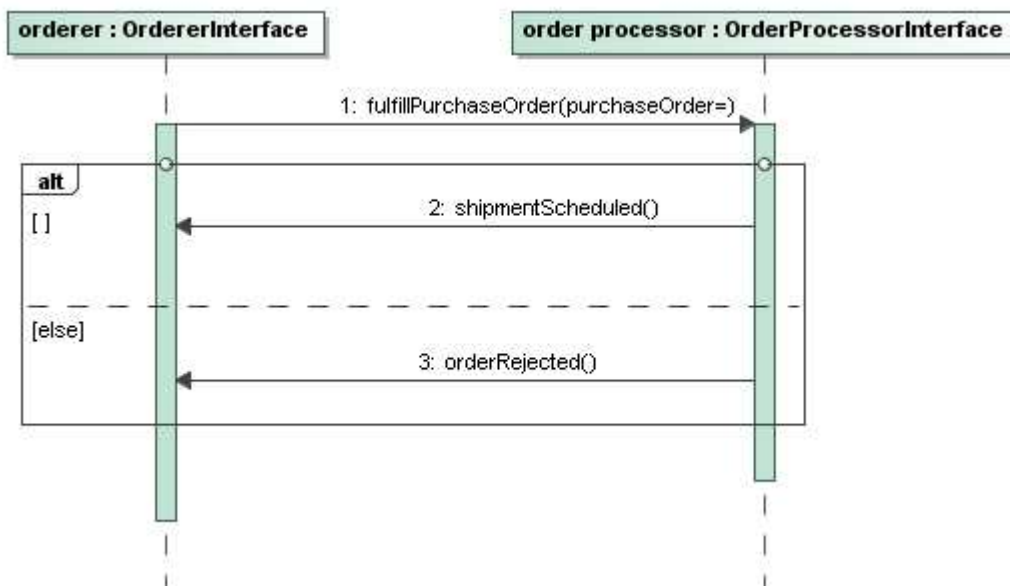


Figure 32: Ordering Service communication protocol

The above behavior (a UML interaction diagram) shows when and what information is transferred between the parties in the service. In this case a fulfillPurchaseOrder message is sent from the orderer to the order processor and the order processor eventually responds with a shipment schedule of an order rejected.

The service interfaces that correspond to the above types are:

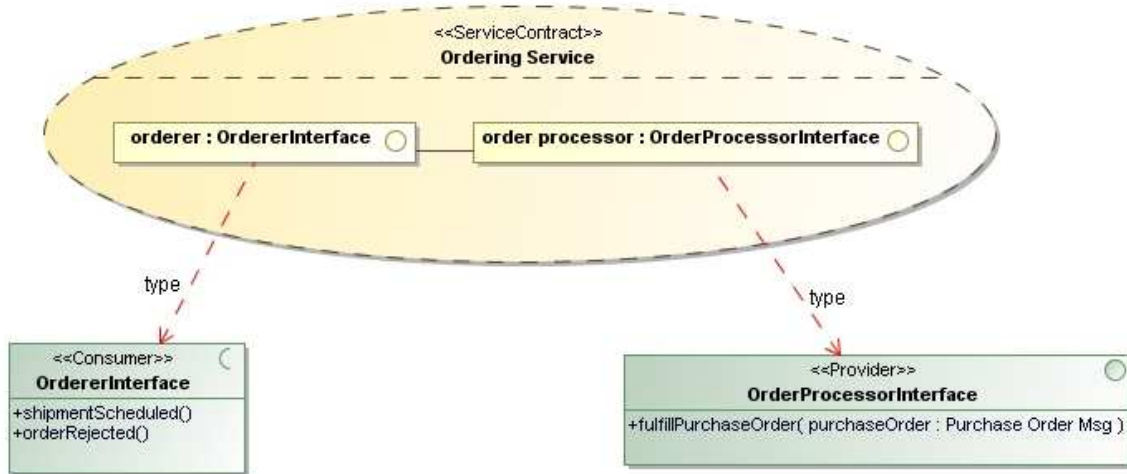


Figure 33: Service interfaces that correspond to the above

As service interface is covered in detail the explanation is not repeated here. Note that the above service interfaces are the types of the roles in the ServiceContract shown in Figure 33.

The following example illustrates compound services:

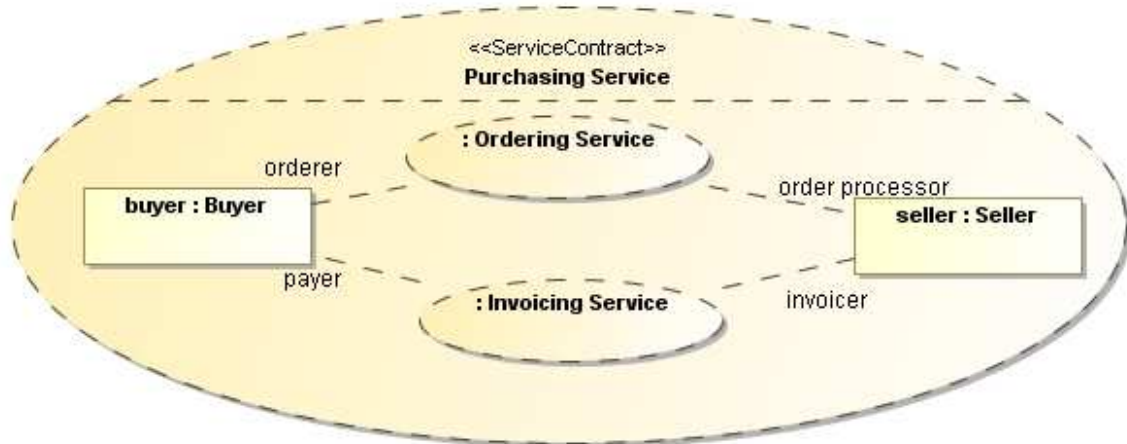


Figure 34: Compound Services

Real-world services are often complex and made up of simpler services as “building blocks”. Using services as building blocks is a good design pattern in that it can decouple finer grain serves and make them reusable across a number of service contracts. Finer grain services may then be delegated to internal actors or components for implementation. Above is an example of a compound ServiceContract composed of other, nested, ServiceContracts. This pattern is common when defining enterprise level ServicesArchitectures – which tend to be more complex and span an extended process lifecycle. The purchasing ServiceContract is composed of 2 more granular ServiceContracts: the “Ordering Service” and the “Invoicing Service”. The buyer is the “orderer” of the ordering service and the “invoice receiver” of the invoicing service. The “Seller” is the “Order processor” of the ordering service and the “invoicer” of the invoicing service. ServiceContracts may be nested to any level using this pattern. The purchasing service defines a new ServiceContract by piecing together these other two services. Note that it is common in a compound service for one role to initiate a sub service but then to be the client of the next – there is no expectation that all the services must go the same direction. This allows for long-lived, rich and asynchronous interactions between participants in a service.

Note: A compound ServiceContract should not be confused with a service that is implemented by calling other services, such as may be specified with a Participant_ServicesArchitecture and/or implemented with BPEL. A compound ServiceContract defines a more granular ServiceContract based on other ServiceContracts.

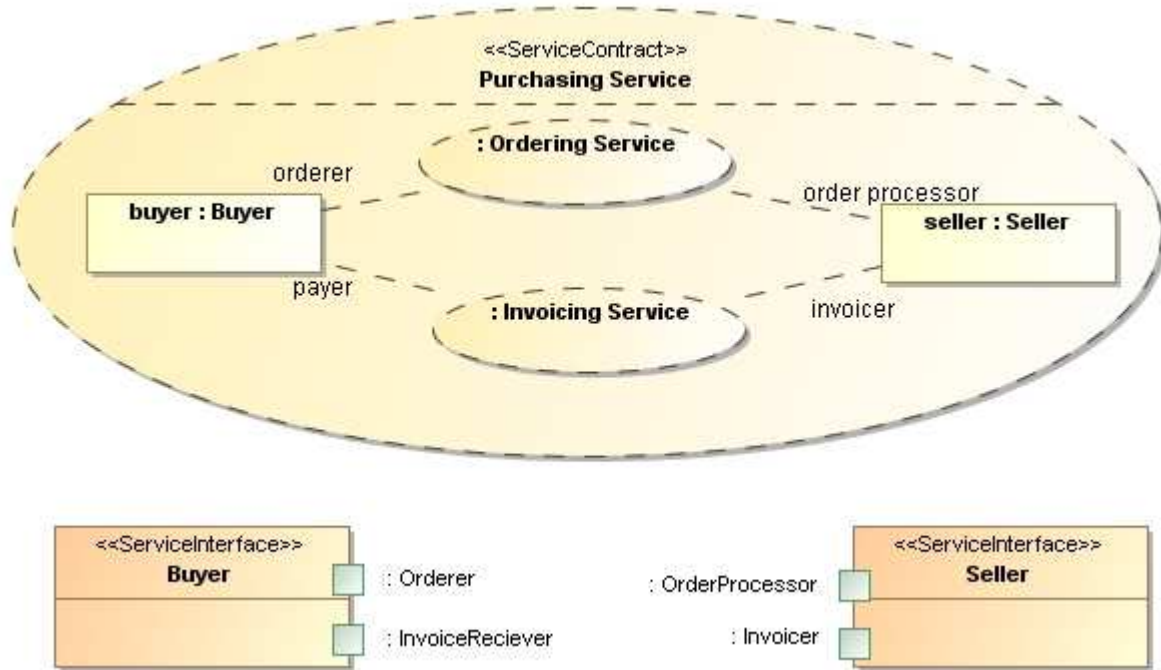


Figure 35: Service Interfaces of a compound service

A compound service has service interfaces with ports, each port representing its role in the larger service contract. The above example shows the Service Interfaces corresponding to the buyer and seller in the purchasing service, a compound service. Note that the seller has two ports, each corresponding to the roles played in the ordering service and invoicing service. Likewise, the buyer has two ports corresponding to the roles it plays in the same services. These ports are typed by the Service Interfaces of the corresponding nested services. The purchasing service specifies how these classes work together and defines the behavioral specification required for each.

When a compound service is used it looks no different than any other service in a services architecture, thus hiding the detail of the more granular service in the high-level architecture yet providing traceability through all levels.

Notation

A ServiceContract is designated using the Collaboration notation stereotyped with «serviceContract» .

Additions to UML2

ServiceContract is a UML collaboration extended as a binding agreement between the parties, designed explicitly to show a service as a contract that is independent of but binding on the involved parties.

ServiceInterface

Defines the interface to a Service Point or Request Point and is the type of a role in a service contract.

Extends Metaclass

Class or Interface

Description

A ServiceInterface defines the interface and responsibilities of a participant to provide or consume a service. It is used as the type of a Service or Request Point. A ServiceInterface is the means for specifying how a participant is to interact to provide or consume a Service. A ServiceInterface may include specific protocols, commands and information exchange by which actions are initiated and the result of the real world effects are made available as specified through the functionality portion of a service. A ServiceInterface may address the concepts associated with ownership, ownership domains, actions communicated between legal peers, trust, business transactions, authority, delegation, etc.

A Service point or Request point or role may be typed by either a ServiceInterface or a simple UML2 Interface. In the latter case, there is no protocol associated with the Service. Consumers simply invoke the operations of the Interface. A ServiceInterface may also specify various protocols for using the functional capabilities defined by the servicer interface. This provides reusable protocol definitions in different Participants providing or consuming the same Service.

A ServiceInterface may specify “parts” and “owned behaviors” to further define the responsibilities of participants in the service. The parts of a ServiceInterface are typed by the Interfaces realized (provided) and used (required) by the ServiceInterface and represent the potential consumers and providers of the functional capabilities defined in those interfaces. The owned behaviors of the ServiceInterface specify how the functional capabilities are to be used by consumers and implemented by providers. A ServiceInterface therefore represents a formal agreement between consumer Requests and providers that may be used to match needs and capabilities.

A service interface may it’s self have service points or request points that define more granular services that serve to make up a larger composite service. This allows “enterprise scale” services to be composed from multiple, smaller services between the same parties. Internal to a participant connections can be made for the entire service or any one of the sub-services, allowing delegation of responsibility for specific parts of the service contract.

A ServiceInterface may fulfill zero or one ServiceContracts by binding the parts of the service contract to the ServiceInterface. Fulfilled contracts may define the functional and nonfunctional requirements for the service interface, the objectives that are intended to be fulfilled by providers of the service, and the value to consumers. In all cases the specification of the ServiceContract and the ServiceInterface may not be in conflict.

Note: There is somewhat of a stylistic difference between specifying service roles and behavior inside of a service interface or in a service contract. In general the service contract is used for more involved services and where a service architecture is being defined, while “standalone” service interfaces may be used for context independent services. However there is some overlap in specification capability and either or both may be used in some cases.

Attributes

No new attributes.

Associations

- No new associations.

Constraints

All parts of a ServiceInterface must be typed by the Interfaces realized or used by the ServiceInterface.

A ServiceInterface must not define the methods for any its provided operations or signals.

Semantics

A ServiceInterface defines a semantic interface to a Service or Request. That is, it defines both the structural and behavioral semantics of the service necessary for consumers to determine if a service typed by a ServiceInterface meets their needs, and for consumers and providers to determine what to do to carry out the service. A ServiceInterface defines the information shown in Table 1.

Function	Metadata
An indication of what the service does or is about	The ServiceInterface name
The service defined by the ServiceInterface that will be provided by any Participant having a Service typed by the ServiceInterface, or used by a Participant having a Request typed by the ServiceInterface	<p>The provided Interfaces containing Operations modeling the capabilities.</p> <p>As in UML2, provided interfaces are designated using an InterfaceRealization between the ServiceInterface and other Interfaces.</p>
Any service interaction consumers are expected to provide or consume in order to use or interact with a Service typed by this ServiceInterface	<p>Required Interfaces containing Operations modeling the needs.</p> <p>As in UML2, required interfaces are designated using a Usage between the ServiceInterface and other Interfaces.</p>
<p>The detailed specification of an interaction providing value as part of a service including:</p> <p>Its name, often a verb phrase indicating what it does</p> <p>Any required or optional service data inputs and outputs</p> <p>Any preconditions consumers are expected to meet before using the capability</p> <p>Any post conditions consumers participants can expect, and other providers must provide upon successful use of the service</p> <p>Any exceptions or fault conditions that might be raised if the capability cannot be provided for some reason even though the preconditions have been met</p>	<p>Each atomic interaction of a ServiceInterface is modeled as an Operation or event reception in its provided or required Interfaces.</p> <p>From UML2, an Operation has Parameters defining its inputs and outputs, preconditions and post-conditions, and may raise Exceptions. Operation Parameters may also be typed by a MessageType.</p>
Any communication protocol or rules that determine when a consumer can use the capabilities or in what order	<p>An ownedBehavior of the ServiceInterface. This behavior expresses the expected interaction between the consumers and providers of services typed by this ServiceInterface.</p> <p>The onwedBehavior could be any Behavior including Activity, Interaction, StateMachine, ProtocolStateMachine, or OpaqueBehavior.</p>

Function	Metadata
Requirements any implementer must meet when providing the service	This is the same ownedBehavior that defines the consumer protocol just viewed from an implementation rather than a usage perspective.
Constraints that reflect what successful use of the service is intended to accomplish and how it would be evaluated	UML2 Constraints in ownedRules of the ServiceInterface.
Policies for using the service such as security and transaction scopes for maintaining integrity or recovering from the inability to successfully perform the service or any required service	Policies may also be expressed as constraints.
Qualities of service consumers should expect and providers are expected to provide such as: cost, availability, performance, footprint, suitability to the task, competitive information, etc.	The OMG QoS specification may be used to model qualities of service constraints for a ServiceInterface.
A service composed of other services as a composite service	Service point or request point ports on the service interface.

Table 1: Information in a ServiceInterface

The semantics of a ServiceInterface are essentially the same as that for a UML2 Class which ServiceInterface specializes. A ServiceInterface formalizes a pattern for using interfaces and classes, and the parts of a class's internal structure to model interfaces to services.

Participants specify their needs with Request points and their capabilities with Service points. Services and Requests, like any part, are described by their type which is either an Interface or a ServiceInterface. A request point may be connected to a compatible Service point in an assembly of Participants through a ServiceChannel. These connected participants are the parts of the internal structure of some other Participant where they are assembled in a context for some purpose, often to implement another service, and often adhering to some ServicesArchitecture. ServiceChannel specifies the rules for compatibility between a Request and Service. Essentially they are compatible if the needs of the Request are met by the capabilities of the Service and they are both structurally and behaviorally compatible.

A ServiceInterface specifies its provided capabilities through InterfaceRealizations. A ServiceInterface can realize any number of Interfaces. Some platform specific models may restrict the number of realized interfaces to at most one. A ServiceInterface specifies its required needs through Usage dependences to Interfaces. These realizations and usages are used to derive the provided and required interfaces of Request and service ports typed by the ServiceInterface.

The parts of a ServiceInterface are typed by the interfaces realized or used by the ServiceInterface. These parts (or roles) may be used in the ownedBehaviors to indicate how potential consumers and providers of the service are expected to interact. A ServiceInterface may specify communication protocols or behavioral rules describing how its capabilities and needs must be used. These protocols may be specified using any UML2 Behavior.

A ServiceInterface may have ownedRules determine the successful accomplishment of its service goals. An ownedRule is a UML constraint within any namespace, such as a ServiceInterface.

Semantic Variation Points

When the ownedRules of a ServiceInterface are evaluated to determine the successful accomplishment of its service goals is a semantic variation point. How the ownedBehaviors of a ServiceInterface are evaluated for conformance with behaviors of consuming and providing Participants is a semantic variation point.

Notation

Denoted using a «serviceInterface» on a Class or Interface.

Examples

Figure 36 shows an example of a simple Interface that can be used to type a Service or Request. This is a common case where there is no required interface and no protocol. Using an Interface as type for a Service point or Request point is similar to using a WSDL PortType or Java interface as the type of an SCA component's service or reference.



Figure 36: The StatusInterface as a simple service interface

Figure 37 shows a more complex ServiceInterface that does involve bi-directional interactions between the parties modeled as provided and required interfaces and a protocol for using the service capabilities. As specified by UML2, Invoicing is the provided interface as derived from the interface realization. InvoiceProcessing is the required interface as derived from the usage dependency.

The invoicing and orderer parts of the ServiceInterface represent the consumer and provider of the service. That is, they represent the Service and Request ports at the endpoints of a ServiceChannel when the service provider is connected to a consumer. These parts are used in the protocol to capture the expected interchange between the consumer and provider.

The protocol for using the capabilities of a service, and for responding to its needs is captured in an ownedBehavior of the ServiceInterface. The invoicingService Activity models the protocol for the InvoicingService. From the protocol we can see that `initiatePriceCalculation` must be invoked on the invoicing part followed by `completePriceCalculation`. Once the price calculation has been completed, the consumer must be prepared to respond to `processInvoice`. It is clear which part represents the consumer and provider by their types. The providing part is typed by the provided interface while the consuming part is typed by the required interface.

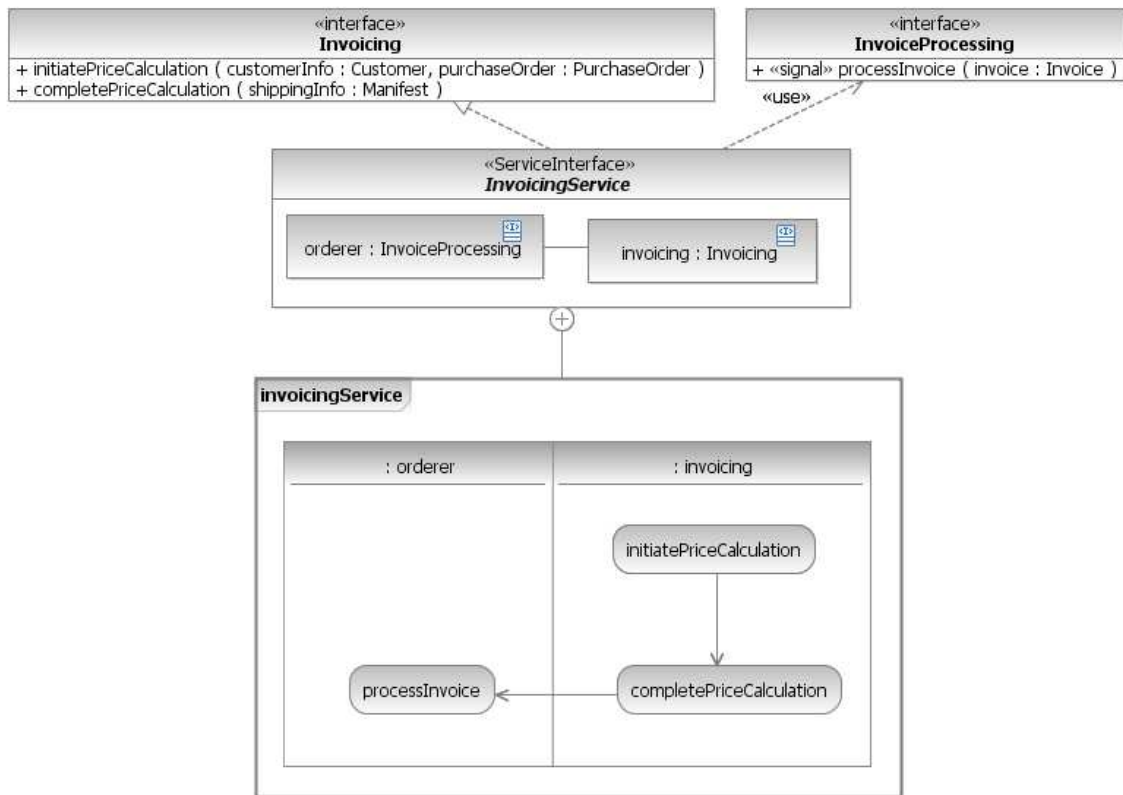


Figure 37: The InvoicingService ServiceInterface

A ServiceInterface may have more than two parts indicating a connector between the consuming and providing ports may have more than two ends, or there may be more than one connection between the ports as specified for UML2. Usually services will be binary, involving just to parties. However, ServiceInterfaces may use more than two parts to provide more flexible allocation of work between consumers but such services may be better specified with a ServiceContract.

Figure 38 shows another version of the ShippingService ServiceInterface that has three parts instead of two. A new part has been introduced representing the scheduler. The orderer part is not typed in the example because it provides no capabilities in the service interface. The protocol indicates that the orderer does not necessarily have to process the schedule; a separate participant can be used instead. This allows the work involved in the ShippingService to be divided among a number of participants.

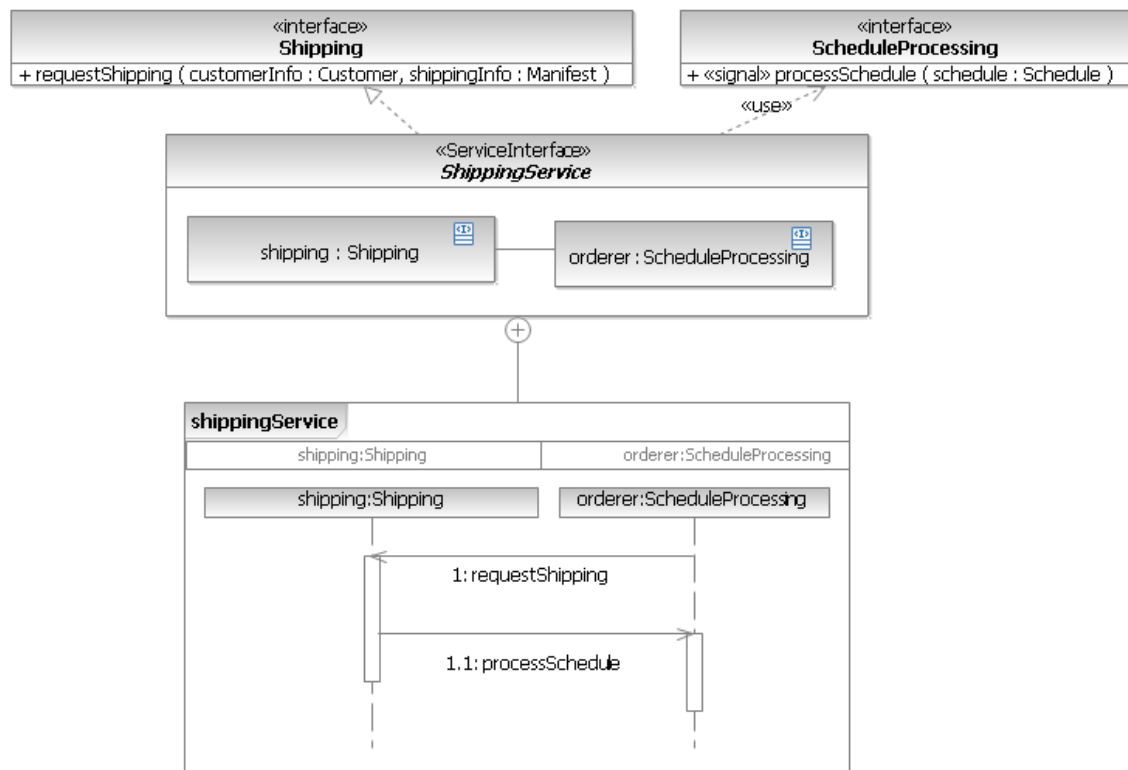


Figure 38: Another version of the ShippingService supporting additional parties

Figure 39 shows an example where aCustomer invokes the requestShipping operation of the shipping service, but aScheduler processes the schedule. This is possible because the ServiceInterface separates the request from the reply by adding the scheduler part. It is the combination of both ServiceChannels that determine compatibility with the shipping service, not just one or the other. That is, it is the combination of all the interactions through a service port that have to be compatible with the port's protocol, not each one.

Figure 40 Shows a different version of the OrderingSubsystem where aCustomer both requests the shipping and processes the order. This ServiceChannel is also valid since this version of aCustomer follows the complete protocol without depending on another part.

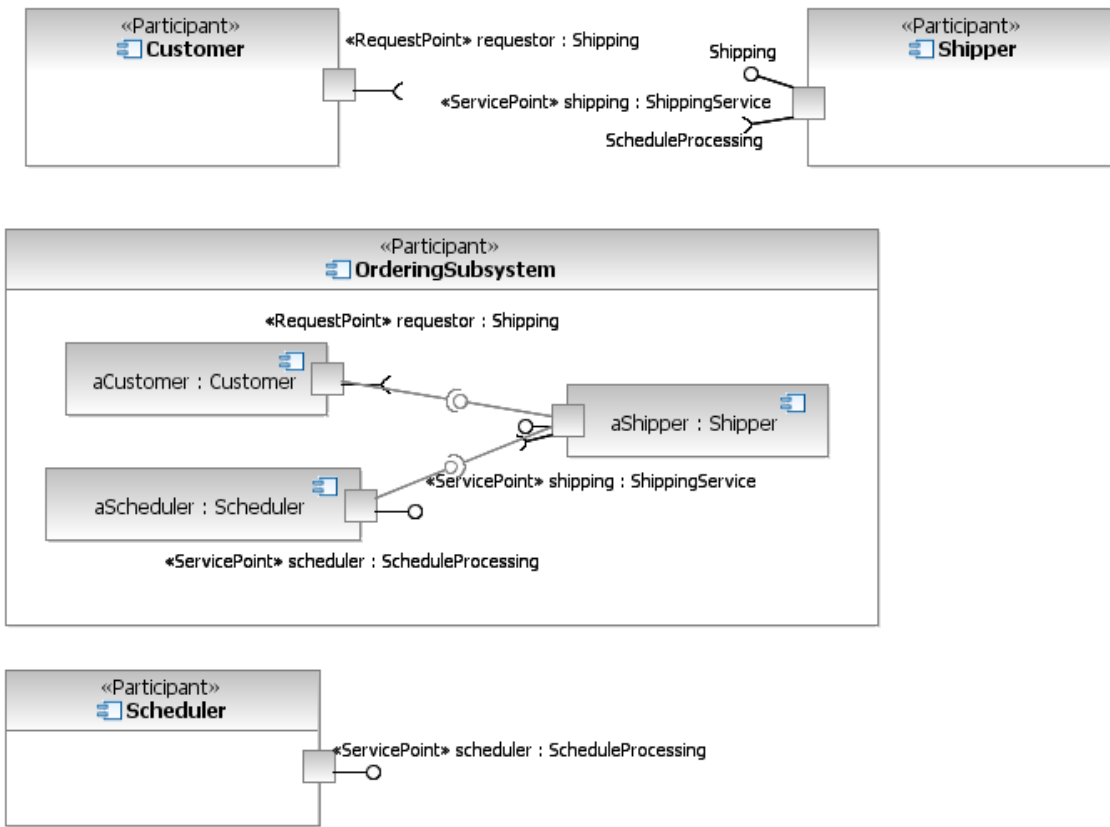


Figure 39: Using the shipping Service with two Consumers

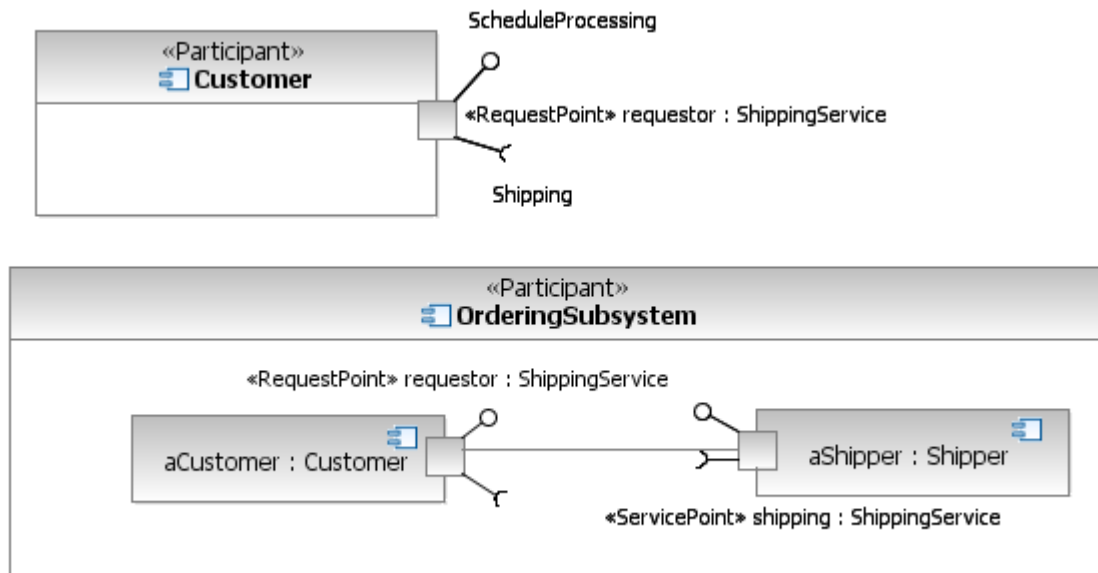


Figure 40: Using the shipping Service with one Consumer

Additions to UML2

Defines the use of a Class or Interface to define the type of a Request or Service point.

ServicesArchitecture

The high-level view of a Service Oriented Architecture that defines how a set of participants works together, forming a community, for some purpose by providing and using services.

Extends Metaclass

Collaboration

Description

A ServicesArchitecture (a SOA) describes how participants work together for a purpose by providing and using services expressed as service contracts. By expressing the use of services, the ServicesArchitecture implies some degree of knowledge of the dependencies between the participants in some context. Each use of a service in a ServicesArchitecture is represented by the use of a ServiceContract bound to the roles of participants in that architecture.

Note that use of a ServicesArchitecture is optional but is recommended to show a high level view of how a set of Participants work together for some purpose. Where as simple services may not have any dependencies or links to a business process, enterprise services can often only be understood in context. The services architecture provides that context—and may also contain a behavior—which is the business process. The participant's roles in a services architecture correspond to the swim lanes or pools in a business process.

A ServicesArchitecture may be specified externally – in a “B2B” type collaboration where there is no controlling entity or as the ServicesArchitecture of a participant - under the control of a specific entity and/or business process. A “B2B” services architecture uses the «servicesArchitecture» stereotype on a collaboration and a participant services architecture uses the «ParticipantArchitecture» stereotype.

A Participant may play a role in any number of services architecture thereby representing the role a participant plays and the requirements that each role places on the participant.

Attributes

No new attributes.

Associations

- No new associations.

Constraints

The parts of a ServicesArchitecture must be typed by a Participant or capability. Each participant satisfying roles in a ServicesArchitecture shall have a port for each role binding attached to that participant. This port shall have a type *compliant* with the type of the role used in the ServiceContract..

Semantics

Standard UML2 Collaboration semantics are augmented with the requirement that each participant used in a services architecture must have a port compliant with the ServiceContracts the participant provides or uses, which is modeled as a role binding to the use of a service contract.

Examples

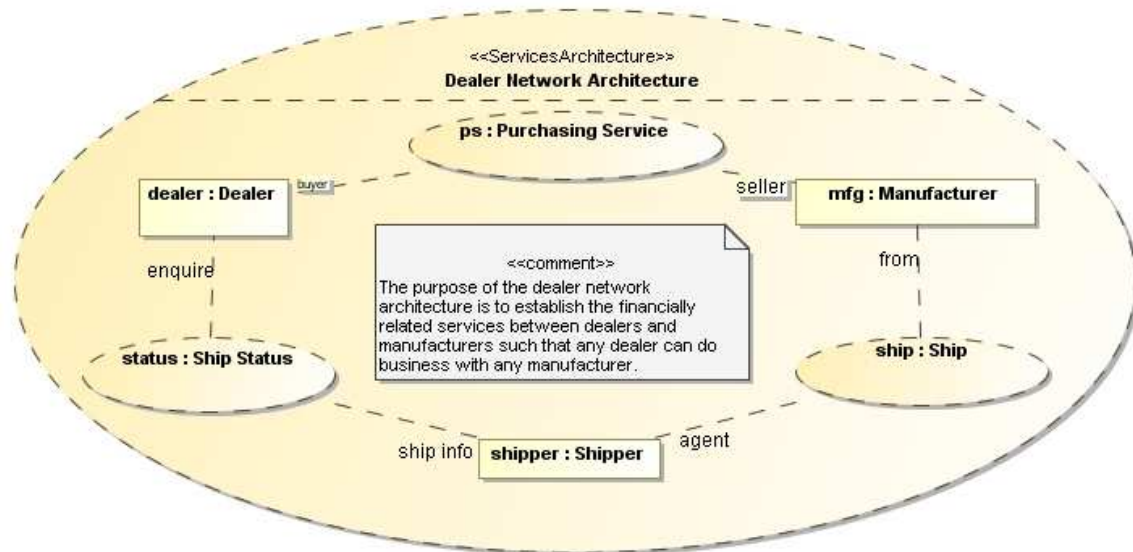


Figure 41: Services architecture involving three participants

The example (Figure 41) illustrates a services architecture involving three participants (dealer, mfg and shipper) and three services (Purchasing Service, Ship Status and Ship). This services architecture shows how a community of dealers, manufacturers and shippers can work together – each party must provide and use the services specified in the architecture. If they do, they will then be able to participate in this community.

This “B2B” SOA specifies the roles of the parties and the services they provide and use without specifying anything about who they are, their organizational structure or internal processes. No “controller” or “mediator” is required as long as each agrees to the service contracts.

By specifying a ServicesArchitecture we can understand the services in our enterprise and communities in context and recognize the real (business) dependencies that exist between the participants. The purpose of the services architecture may also be specified as a comment.

Each participant in a ServicesArchitecture must have a port that is compatible with the roles played in each ServiceContract role it is bound to.

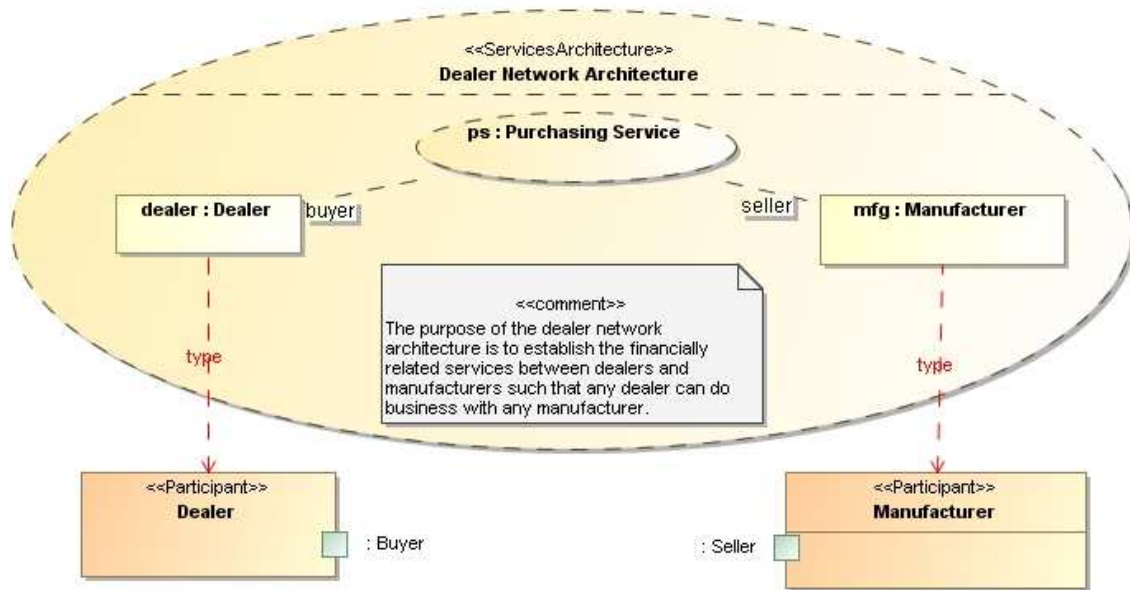


Figure 42: Abbreviated service contract

The diagram in Figure 42 depicts an abbreviated service contract with the participant types and their ports (the red dependencies are illustrative and show the type of the roles). Note that the participants each have a port corresponding to the services they participate in.