Date: May 2012

Service oriented architecture Modeling Language (SoaML) Specification

Version 1.0.1

Normative reference: http://www.omg.org/spec/SoaML/1.0.1
Machine consumable files: http://www.omg.org/spec/SoaML/20120501
http://www.omg.org/spec/SoaML/20120501/SoaMLMetamodel.xmi
http://www.omg.org/spec/SoaML/20120501/SoaMLProfile.xmi

Copyright © 2008, Adaptive Ltd.
Copyright © 2008, Capgemini
Copyright © 2008, CSC
Copyright © 2008, EDS
Copyright © 2008, Fujitsu
Copyright © 2008, Fundacion European Software Institute
Copyright © 2008, Hewlett-Packard
Copyright © 2008, International Business Machines Corporation
Copyright © 2008, MEGA International
Copyright © 2008, Model Driven Solutions
Copyright © 2012, Object Management Group
Copyright © 2008, Rhysome
Copyright © 2008, Softeam

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , IMM™ , MOF™ , OMG Interface Definition Language (OMG IDL)™ , and OMG Systems Modeling Language (OMG SysML)™ 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.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

March 2012:OMG internal document number: formal/2012 -03-01

May 2012:OMG internal document number: formal/2012-05-10 due to version string being updated to 20120501.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

Preface v

1 Scope 1

2 Conformance 1

3 Normative References 3

4 Terms and Definitions (informative) 4

5 Additional Information 5

5.1 How to Read this Specification 5

5.2 Acknowledgements 5

6 SoaML UML Profile 7

6.1 Introduction to SoaML 7

6.1.1 On Service and Service Oriented Architecture (SOA) 7

6.1.2 Supporting both an IT and a Business Perspective on SOA 7

6.1.3 Supporting both a Contract and an interface based approach to SOA 8

6.1.4 Supporting both Top down and bottom-up development for SOA 9

6.1.5 Key Concepts of Basic Services 10

6.2 Use of the Simple Interface to Define Participants 11

6.2.1 Interface based SOA 12

6.2.1.1 Specifying the choreography 14

6.2.1.2 Use of the service interface to define Participants 14

6.2.2 Key Concepts of the Services Architecture 15

6.2.3 Service Contracts and Contract based SOA 17

6.2.4 Example of Contract based SOA 20

6.2.4.1 Use of the service contract to define participants 22

6.2.4.2 Use of “Service” and “Request” (Alternative) 22

6.2.4.3 Multi-Party Service Contracts 23

6.2.4.4 Participant ports in support of the multi-party service contract 25

6.2.4.5 Adapting Existing Services 26

6.2.4.6 Defining a ServiceContract for a simple interface 26

6.2.4.7 Showing how participants work together 27

6.2.4.8 Services Architecture for a Participant 27

6.2.5 Capability 29

6.2.5.1 Capabilities represent an abstraction of the ability to affect change 29

6.2.6 Business Motivation 32

6.3 The SoaML Profile of UML 33

6.4 Stereotype Descriptions 35
6.4.1 Agent 35
6.4.2 Attachment 38
6.4.3 Capability 39
6.4.4 Consumer 40
6.4.5 Collaboration 42
6.4.6 CollaborationUse 43
6.4.7 Expose 46
6.4.8 MessageType 47
6.4.9 Milestone 50
6.4.10 Participant 52
6.4.11 Port 56
6.4.12 Property 57
6.4.13 Provider 59
6.4.14 Request 60
6.4.15 ServiceChannel 62
6.4.16 ServiceContract 65
6.4.17 ServiceInterface 70
6.4.17.1 Semantic Variation Points 73
6.4.18 Service 77
6.4.19 ServicesArchitecture 80
7 Categorization 83
7.1 Overview 83
7.2 Abstract Syntax 83
7.3 Class Descriptions 84
7.3.1 Catalog 84
7.3.2 Categorization 85
7.3.3 Category 86
7.3.4 CategoryValue 88
7.3.5 RAS Placeholders 88
8 BMM Integration 91
8.1 Overview 91
8.2 Abstract Syntax 91
8.3 Class and Stereotype Descriptions 91
8.3.1 MotivationElement 91
8.3.2 MotivationRealization 92
9 SoaML Metamodel 95

9.1 Overview 95
9.2 Profile metamodel mapping 98

Annex A: Relationship to OASIS Services Reference Model 99
Annex B: Examples 105
Annex C: Purchase Order Example with Fujitsu SDAS/SOA 127
Annex D: BPDM Mapping 131

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

OMG Domain specifications

OMG Embedded Intelligence Specifications

OMG Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/technology/agreement.htm*.

1 Scope

The Service oriented architecture Modeling Language (SoaML) specification provides a metamodel and a UML profile for the specification and design of services within a service-oriented architecture.

SoaML meets the mandatory requirements of the UPMS (UML Profile and Metamodel for Services) RFP, OMG document number soa/2006-09-09, as described in part I. It covers extensions to UML2.1 to support the following new modeling capabilities:

- Identifying services, the requirements they are intended to fulfill, and the anticipated dependencies between them.
- Specifying services including the functional capabilities they provide, what capabilities consumers are expected to provide, the protocols or rules for using them, and the service information exchanged between consumers and providers.
- Defining service consumers and providers, what requisition and services they consume and provide, how they are connected and how the service functional capabilities are used by consumers and implemented by providers in a manner consistent with both the service specification protocols and fulfilled requirements.
- The policies for using and providing services.
- The ability to define classification schemes having aspects to support a broad range of architectural, organizational, and physical partitioning schemes and constraints.
- Defining service and service usage requirements and linking them to related OMG metamodels, such as the BMM course_of_action, BPDM Process, UPDM OperationalCapability, and/or UML UseCase model elements they realize, support, or fulfill.
- SoaML focuses on the basic service modeling concepts, and the intention is to use this as a foundation for further extensions both related to integration with other OMG metamodels like BPDM and BPMN 2.0, as well as SBVR, OSM, ODM, and others.

The rest of this document provides an introduction to the SoaML UML profile and the SoaML Metamodel, with supporting non-normative annexes.

The SoaML specification contains both a SoaML metamodel and a SoaML UML profile. The UML profile provides the flexibility for tool vendors having existing UML2 tools to be able to effectively develop, transform, and exchange services metamodels in a standard way. At the same time it provides a foundation for new tools that wish to extend UML2 to support services modeling in a first class way. The fact that both approaches capture the same semantics will facilitate migration between profile-based services models, and future metamodel extensions to SoaML that may be difficult to capture in profiles.

2 Conformance

Compliance with SoaML requires that the L3 compliance level of UML is implemented, and the extensions to the UML L3 required for SoaML are implemented as described in this specification. Tools may implement SoaML using either the profile or by extending the UML metamodel using package merge and the SoaML metamodel. Either the profile or UML extended with package merge shall comprise compliant SoaML abstract syntax that will be exchanged using the XMI format defined by the UML metamodel and SoaML profile. Tools may optionally also support the XMI as defined by the SoaML metamodel extension and described under compliance levels L2 and L3.

To fully comply with SoaML, a design tool must implement both the concrete syntax (notation) and abstract syntax of SoaML as represented by the XMI exchange format. A SoaML runtime or MDA provisioning tool must implement the XMI exchange format. A design tool is software that provides for graphical modeling. A runtime or provisioning tool is one that uses the SoaML XML exchange format for execution or the generation of other artifacts.

Compliance Levels for SoaML

The following compliance points elaborate the definition of compliance for SoaML:

- SoaML:P1 (Profile compliant) - This compliance point requires implementation of the SoaML UML profile and exchange of XMI as described by UML L3 and the SoaML profile extensions. P1 compliance is intended for extension of “off the shelf” UML tools. The stereotypes included in the Categorization chapter are optional in compliance point P1.
- SoaML:P2 (Metamodel compliant, basic) - This level provides the SoaML meta model as an extension to the core UML abstract syntax contained in the UML L3 package and adds the capabilities provided by the SoaML Services package (Figure 2.1).

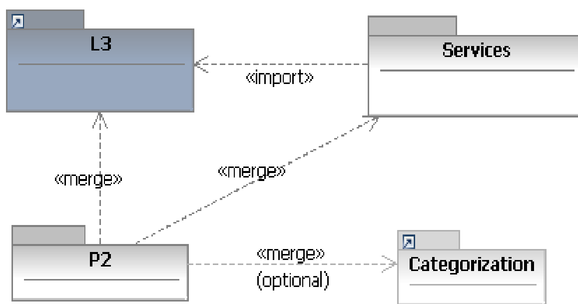


Figure 2.1 - SoaML Compliance Point P2

- SoaML:P3 (Metamodel compliant, with BMM) - This level extends the SoaML provided in P2 and adds in integration with the OMG Business Motivation Model (BMM) (Figure 2.2).

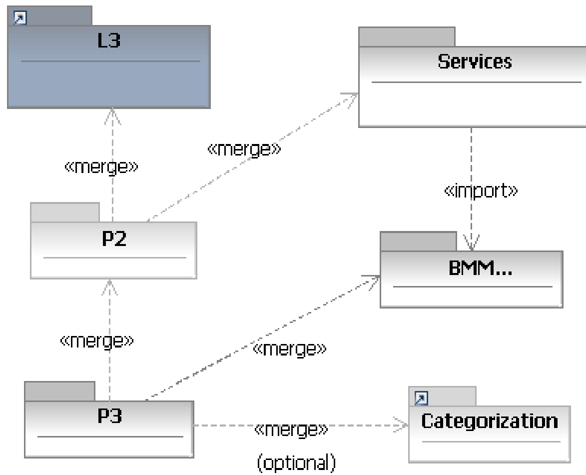


Figure 2.2 - SoaML Compliance Point P3

For all compliance points, package Categorization optionally may be included through inclusion in the SoaML profile, applied as a separate additional profile, or merged into a compliance point of the SoaML metamodel.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. Refer to the OMG site for subsequent amendments to, or revisions of, any of these publications.

- Object Constraint Language, v2.0: OMG document number formal/2006-05-01
- UML Profile and Metamodel for Services RFP: OMG document number soa/06-09-09
- UML, Superstructure, v2.1.2: OMG document number formal/2007-11-02
- UML Infrastructure, v2.1.2: OMG document number formal/2007-11-04
- MOF, v2.0: Omg document number formal/2006-01-01
- MOF 2.0/XMI Mapping, v2.1.1: OMG document number formal/2007-12-01
- UML Profile for Modeling QoS and Fault Tolerance, v1.1: OMG document number formal/2008-04-05
- Business Process Definition Metamodel, v1.0: volume 1: OMG document number formal/2008-11-03 and volume 2: OMG document number formal/2008-11-04
- Business Motivation Model, v1.0: OMG document number formal/2008-08-02
- Ontology Definition Metamodel, v1.0: OMG document number formal/2009-05-01

4 Terms and Definitions (informative)

The terms and definitions are referred to in the SoaML specification and are derived from multiple sources included in the Normative References section of this document.

Meta-Object Facility (MOF)

The Meta Object Facility (MOF), an adopted OMG standard, provides a metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems. Examples of these systems that use MOF include modeling and development tools, data warehouse systems, metadata repositories, etc.

Object Constraint Language (OCL)

The Object Constraint Language (OCL), an adopted OMG standard, is a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Note that when the OCL expressions are evaluated, they do not have side effects; i.e., their evaluation cannot alter the state of the corresponding executing system.

Ontology Definition Metamodel (ODM)

The Ontology Definition Metamodel (ODM), as defined in this specification, is a family of MOF metamodels, mappings between those metamodels as well as mappings to and from UML, and a set of profiles that enable ontology modeling through the use of UML-based tools. The metamodels that comprise the ODM reflect the abstract syntax of several standard knowledge representation and conceptual modeling languages that have either been recently adopted by other international standards bodies (e.g., RDF and OWL by the W3C), are in the process of being adopted (e.g., Common Logic and Topic Maps by the ISO), or are considered industry de facto standards (non-normative ER and DL appendices).

Platform Independent Model (PIM)

A *platform independent model* is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type. Examples of platforms range from virtual machines, to programming languages, to deployment platforms, to applications, depending on the perspective of the modeler and application being modeled.

Platform Specific Model (PSM)

A *platform specific model* is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform.

Unified Modeling Language (UML)

The Unified Modeling Language, an adopted OMG standard, is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecommunications, aerospace) and implementation platforms (e.g., JEE, .NET).

XML Metadata Interchange (XMI)

XMI is a widely used interchange format for sharing objects using XML. Sharing objects in XML is a comprehensive solution that builds on sharing data with XML. XMI is applicable to a wide variety of objects: analysis (UML), software (Java, C++), components (EJB, IDL, CORBA Component Model), and databases (CWM).

eXtensible Markup Language (XML)

Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. RDF and OWL build on XML as a basis for representing business semantics on the Web. Relevant W3C recommendations are cited in the RDF and OWL documents as well as those cited under Normative References, above.

5 Additional Information

5.1 How to Read this Specification

The rest of this document contains the technical content of this specification. As background for this specification, readers are encouraged to first have some general background on service oriented architectures and on UML. See for instance the SOA reference models referred to in Annex B, and the OMG UML specifications. For UML read the *UML: Superstructure* specification that this specification extends. Part I, “Introduction” of *UML: Superstructure* explains the language architecture structure and the formal approach used for its specification. Afterwards the reader may choose to either explore the InfrastructureLibrary, described in Part II, “Infrastructure Library,” or the Classes::Kernel package which reuses it, described in Chapter 1, “Classes.” The former specifies the flexible metamodel library that is reused by the latter; the latter defines the basic constructs used to define the UML metamodel.

Although the chapters are organized in a logical manner and can be read sequentially, this is a reference specification and is intended to be read in a non-sequential manner. Consequently, extensive cross-references are provided to facilitate browsing and search.

5.2 Acknowledgements

The following companies submitted and/or supported parts of this specification:

Submitters

- Adaptive
- Capgemini
- CSC
- EDS
- Fujitsu
- Fundacion European Software Institute
- Hewlett-Packard
- International Business Machines
- MEGA International
- Model Driven Solutions
- Rhysome

- Softeam

Supporters

- BAE Systems
- DERI – University of Innsbruck
- DFKI
- Everware-CBDI
- France Telecom R&D
- General Services Administration
- MID GmbH
- NKUA – University of Athens
- Oslo Software
- SINTEF
- THALES Group
- University of Augsburg
- Wilton Consulting Group

In particular we would like to acknowledge the participation and contribution from the following individuals: Jim Amsden, George Athanasopoulos, Irv Badr, Bernhard Bauer, Mariano Belaunde, Gorka Benguria, Arne J. Berre, John Butler, Cory Casanave, Bob Covington, Fred Cummins, Philippe Desfray, Andreas Ditze, Jeff Estefan, Klaus Fischer, Christian Hahn, Øystein Haugen, PJ Hinton, Henk Kolk, Xabier Larrucea, Jérome Lenoir, Antoine Lonjon, Saber Mansour, Hiroshi Miyazaki, Jishnu Mukerji, James Odell, Michael Pantazoglou, Pete Rivett, Dimitru Roman, Mike Rosen, Stephen Roser, Omair Shafiq, Ed Seidewitz, Bran Selic, Aphrodite Tsalgatidou, Kenn Hussey, Fred Mervine.

6 SoaML UML Profile

6.1 Introduction to SoaML

6.1.1 On Service and Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) is a way of describing 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 value delivered 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 MOF and 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 modeling capabilities.

6.1.2 Supporting both an IT and a Business Perspective on 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. Such technologies may also be supported using SoaML through technology profiles and various “Model Driven Architecture” methods for implementing solutions based on models.

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 can leverage Model Driven Architecture® (MDA®) to help map business and systems architectures, the design of the enterprise, to the technologies that support business automation, 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.

6.1.3 Supporting both a Contract and an interface based approach to SOA

SoaML integrates modeling capabilities in support of using SOA at different levels and with different methodologies. In particular support for a "contract based" and "interface based" approach which, in general, follow the "ServiceContract" and "ServiceInterface" elements of the SoaML profile, respectively.

SoaML supports different approaches to SOA, which results in different, but overlapping profile elements. Before addressing the differences, let's review some of the similarities and terminology.

- Participants - participants are either specific entities or kinds of entities that provide or use services. Participants can represent people, organizations, or information system components. Participants may provide any number of services and may consume any number of services.
- Ports - participants provide or consume services via ports. A port is the part or feature of a participant that is the interaction point for a service - where it is provided or consumed. A port where a service is offered may be designated as a "Service" port and the port where a service is consumed may be designated as a "Request" port.
- Service description - the description of how the participant interacts to provide or use a service is encapsulated in a specification for the service - there are three ways to specify a service interaction - a UML Interface, a ServiceInterface, and a ServiceContract. These different ways to specify a service relate to the SOA approach and the complexity of the service, but in each case they result in interfaces and behaviors that define how the participant will provide or use a service through ports. The service descriptions are independent of, but consistent with how the provider provides the service or how (or why) the consumer consumes it. This separation of concerns between the service description and how it is implemented is fundamental to SOA. A service specification specifies how consumers and providers are expected to interact through their ports to enact a service, but not how they do it.
- Capabilities - participants that provide a service must have a capability to provide it, but different providers may have different capabilities to provide the same service - some may even "outsource" or delegate the service implementation through a request for services from others. The capability behind the service will provide the service according to a service description and may also have dependencies on other services to provide that capability. The service capability is frequently integral to the provider's business process. Capabilities can be seen from two perspectives, capabilities that a participant has that can be exploited to provide services, and capabilities that an enterprise needs that can be used to identify candidate services.

These approaches to specifying a service can be summarized as:

- Simple Interfaces - The simple interface focuses attention on a one-way interaction provided by a participant on a port represented as a UML interface. The participant receives operations on this port and may provide results to the caller. This kind of one-way interface can be used with "anonymous" callers and the participant makes no assumptions about the caller or the choreography of the service. The one-way service corresponds most directly to simpler "RPC style web services" as well as many "OO" programming language objects. A simple interface used to type a service port can optionally be a ServiceInterface. Simple interfaces are often used to expose the "raw" capability of existing systems or to define simpler services that have no protocols. Simple interfaces are the degenerate case of both the ServiceInterface and ServiceContract where the service is unidirectional - the consumer calls operations on the provider - the provider doesn't callback the consumer and may not even know who the consumer is.
- ServiceInterface based - A ServiceInterface based approach allows for bi-directional services, those where there are "callbacks" from the provider to the consumer as part of a conversation between the parties. A service interface is defined in terms of the provider of the service and specifies the interface that the provider offers as well as the interface, if any, it expects from the consumer. The service interface may also specify the choreography of the service - what information is sent between the provider and consumer and in what order. A consumer of a service specifies the service interface they require using a request port. The provider and consumer interfaces must either be the same or compatible. If they are compatible, the provider can provide the service to that consumer. The consumer must adhere

to the provider's service interface, but there may not be any prior agreement between the provider and consumer of a service. Compatibility of service interfaces determines whether these agreements are consistent and can therefore be connected to accomplish the real world effect of the service and any exchange in value. The ServiceInterface is the type of a "Service" port on a provider and the type of a "Request" port on the consumer. In summary, the consumer agrees to use the service as defined by its service interface, and the provider agrees to provide the service according to its service interface. Compatibility of service interfaces determines whether these agreements are consistent and can therefore be connected. The ServiceInterface approach is most applicable where existing capabilities are directly exposed as services and then used in various ways, or in situations that involve one or two parties in the service protocol.

- ServiceContract based - A service contract approach defines service specifications (the service contract) that specify how providers, consumers and (potentially) other roles work together to exchange value. The exchange of value is the enactment of the service. The service contract approach defines the roles each participant plays in the service (such as provider and consumer) and the interfaces they implement to play that role in that service. These interfaces are then the types of ports on the participant, which obligates the participant to be able to play that role in that service contract. The service contract represents an agreement between the parties for how the service is to be provided and consumed. This agreement includes the interfaces, choreography, and any other terms and conditions. Note that the agreement may be asserted in advance or arrived at dynamically, as long as an agreement exists by the time the service in enacted. If a provider and consumer support different service contracts, there must be an agreement or determination that their different service contracts are compatible and consistent with other commitments of the same participants. Service contracts are frequently part of one or more services architectures that define how a set of participants provide and use services for a particular business purpose or process. In summary, the service contract approach is based on defining the service contract "in the middle" (between the parties) and having the ends (the participants) agree to their part in that contract, or adapt to it. The ServiceContract approach is most applicable where an enterprise, community SOA architecture or choreography is defined and then services built or adapted to work within that architecture, or when there are more than two parties involved in the service.

The fundamental differences between the contract and interface based approaches is whether the interaction between participants are defined separately from the participants in a ServiceContract that defines the obligations of all the participants, or individually on each participants' service and request.

6.1.4 Supporting both Top down and bottom-up development for SOA

SoaML can be used for basic context independent services like the "Get stock quote" or "get time" examples popular in web services. 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 agreements captured in the services architecture of that community. A SoaML ServicesArchitecture 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 eventually 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 capabilities arranged in a hierarchy showing anticipated usage dependencies and using these capabilities to identify service interfaces that expose them through services.

- 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.
- Identifying services from existing assets that can be used by participants to adapt those capabilities and expose them as services.
- Identifying common data and data flows between parties and grouping these into services.

Regardless of how services are identified, they are formalized by service descriptions. A service description defines the purpose of the service and any interaction or communication protocol for how to properly use and provide a service. A service description 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 the services, and possibly using other services. Services implementations may be specified by methods that are 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 solution, perhaps specified by, and adhering to 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.

6.1.5 Key Concepts of Basic Services

A key concept is, of course, service. Service is defined as the delivery 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]

As mentioned earlier, provider and consumer entities may be people, organizations, technology components or systems - we call these Participants. Participants offer services through Ports that may use the "Service" stereotype and request services on Ports with the "Request" stereotype. These ports represent features of the participants where the service is offered or consumed.

The service port 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 port 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 6.1 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 port specifies that participant

“Productions” has the capability to offer that service, which could, for example, be described in a UDDI repository. Note that a participant may also offer other services on other service ports. Participant “Productions” has two owned behaviors that are the methods of the operations provided through the scheduling service.

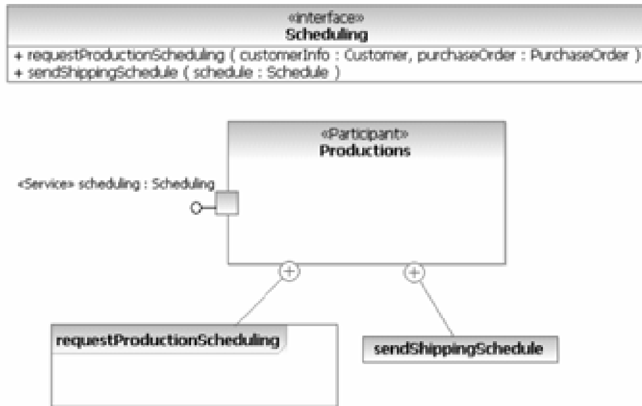


Figure 6.1 - Services and Service Participants

6.2 Use of the Simple Interface to Define Participants

Simple interfaces define one-way services that do not require a protocol. Such services may be defined with only a single UML interface and then provided on a “Service” port and consumed on a “Request” port.

Figure 6.2 - Simple Interface

The above interface fully defines the service. It could, of course, optionally be used in a ServiceInterface or ServiceContract – but this is not required.

The UML interface may be used as the type of «Service» ports and «Request» ports.

Figure 6.3 - Use of simple interface

The above diagram shows the use of “Shipment status” as the type of a «Service» port and «Request» port. When used in the «Service» port the interface is provided. When used in the «Request» port the service is used and the resulting ports are compatible.

6.2.1 Interface based SOA

Like a UML interface, a ServiceInterface defines or specifies a service and can be the type of a service port. The service interface has the additional feature that it can specify a bi-directional service having a protocol – where both the provider and consumer have responsibilities to invoke and respond to operations, send and receive messages or 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 value provided, 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 value required, 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, constraining but 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 Service port or Request port (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 of the service interface.
- 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 or are made available through 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 that service providers need to follow in order to implement the service.

The focus of an interface based SOA is the specification of ServiceInterfaces provided or used by a participant's ports. The ServiceInterface specification of these ports fully specifies the participant's requirements to provide the service on a «Service» or to request a service on a «Request». The ports are then compatible if the service interfaces are compatible. A ServiceInterface is not a UML interface, but a class providing and requiring the interfaces of the provider.

A ServiceInterface is a UML Class and defines specific roles each participant plays in the service interaction. These roles have a name and an interface type. The interface of the provider (which must be the type of one of the parts in the class) is realized (provided) by the ServiceInterface class. The interface of the consumer (if any) must be used by the class. This example demonstrates such a service interface.

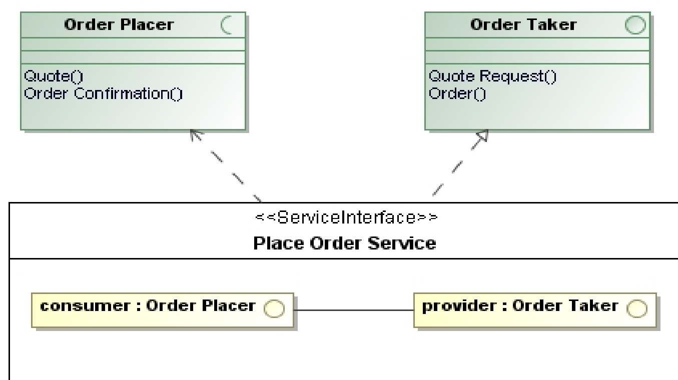


Figure 6.4 - Service Interface Definition

Let’s look at each part individually:

- **«ServiceInterface»Place Order Service** – this is the root of the service interface and represents the service itself – the terms and conditions under which the service can be enacted and the results of the service. The service interface may be related to business goals or requirements. The service interface can also be used in services architectures to show how multiple services and participants work together for a business purpose. This service interface is defined from the perspective of the provider of the service – the order taker.
- **provider : Order Taker** – this defines the role of the provider in the place order service. The provider is the participant that provides something of value to the consumer. The type of the provider role is “Order Taker,” this is the interface that a provider will require on a port to provide this service.
- **consumer: Order Placer** – this is the role of the consumer in the place order service. The consumer is the participant that has some need and requests a service of a provider. The type of the consumer role is “Order Placer,” this is the interface that a consumer will implement on a port to consume this service (note that in the case of a one-directional service, there may not be a consumer interface).
- **Order Taker** – this is the interface for a place order service provider. This indicates all the operations and signals a providing participant may receive when enacting this service. Note that the Place Order Service provides the same interface that makes this service interface provider centric.
- **Order Placer –** this is the interface for a place order service consumer. This indicates all of the operations and signals a consuming participant will receive when enacting the service. In simple unidirectional services, the consumer interface may be missing or empty.

6.2.1.1 Specifying the choreography

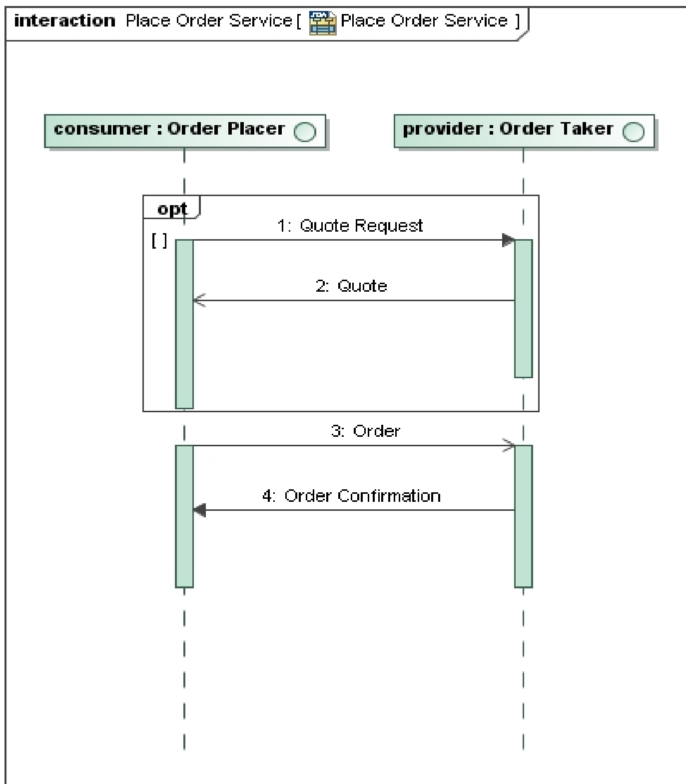


Figure 6.5 - Place order choreography

The service choreography, above, is a behavior owned by the service interface and defines the required and optional interactions between the provider and consumer. There are two primary interaction sets - the quote request resulting in a quote and the order resulting in an order confirmation. In this case these interactions can be defined using signals, which makes this service interface asynchronous and document oriented, a mainstream SOA best practice.

6.2.1.2 Use of the service interface to define Participants

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, in Figure 6-6, the figure to the right, illustrates a Manufacturer participant that offers a purchaser service. Participants provide capabilities through Service ports 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 feature or interaction point through which a classifier interacts with other classifiers (see Figure 6.6). A port typed by a ServiceInterface and designated with the «Service» stereotype is known as a *service port*. A *service port* is the feature that represents the point of interaction on a Participant where a service is actually provided. 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 port* is the point of interaction for engaging participants in a service via its service interfaces.

Just as we want to define the services provided by a participant using a service port, 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 «Request» port.

The ServiceInterface is used to type «Service» ports and «Request» ports of participants. The provider of the service uses a «Service» port and the consumer of the service uses a «Request» port. The «Service» ports and «Request» ports are the points of interaction for the service. Let’s look at some participants:

Figure 6.6 - Participating in services

Note that both the dealer and manufacturer have a port typed with the “Place order service.” The manufacturer is the provider of the place order service and has a «Service» port. The dealer is a consumer of the place order service and uses a «Request» port. Note that the manufacturer’s port provides the “Order Taker” interface and requires the “Order Placer” interface.

Since the dealer uses a «Request» the “conjugate” interfaces are used – so the dealer’s port provides the Order Placer interfaces and uses the Order Taker. Since they are conjugate, the ports on the dealer and manufacturer can be connected to enact the service. When «Request» is used the type name will be preceded with a tilde (“~”) to show that the conjugate type is being used.

Showing the type and interfaces is a bit overkill, so we usually only show the port type on diagrams – but the full UML notation should help to clarify how ports are used to provide and consume services. Note that these participants have other ports, showing that it is common for a participant to be the provider and consumer of many services.

6.2.2 Key Concepts of the Services Architecture

One of the key benefits of SOA is the ability to enable a community, organization, or system of systems 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 to support the goals community, system of systems, 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, component 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 an architecture – one that specifies how parts of that participant (e.g., departments within an organization) work together to provide the services of the owning participant. The architecture of a participant is shown as a Services Architecture as well, perhaps realized by a UML class or component and frequently has an associated business process. Participants that realize this specification must adhere to the architecture it specifies.

A ServicesArchitecture (see Figure 6.7) is defined using a UML Collaboration.

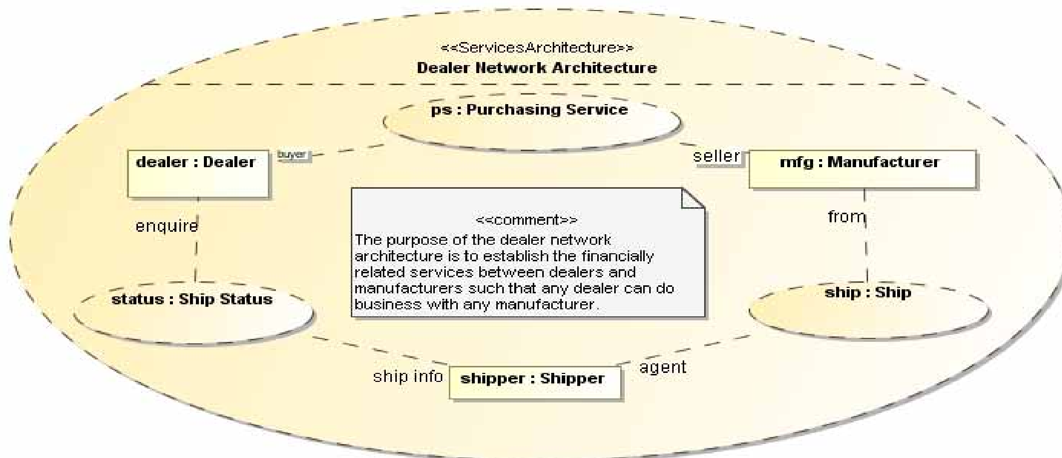


Figure 6.7 - Example community services architecture with participant roles and services

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 ports required of the entities that fill these roles and are then bound by the services architectures in which they participate.

A ServicesArchitecture can be used to specify the architecture for a particular Participant. Within a participant, where there is a concept of “management” exists, a services architecture illustrates how realizing participants and external collaborators work together and would often be accompanied by a business process. A ServicesArchitecture or specification class may be composed from other services architectures and service contracts. As shown in Figure 6.7, 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 ports that have a type compatible with the services they must provide and consume.

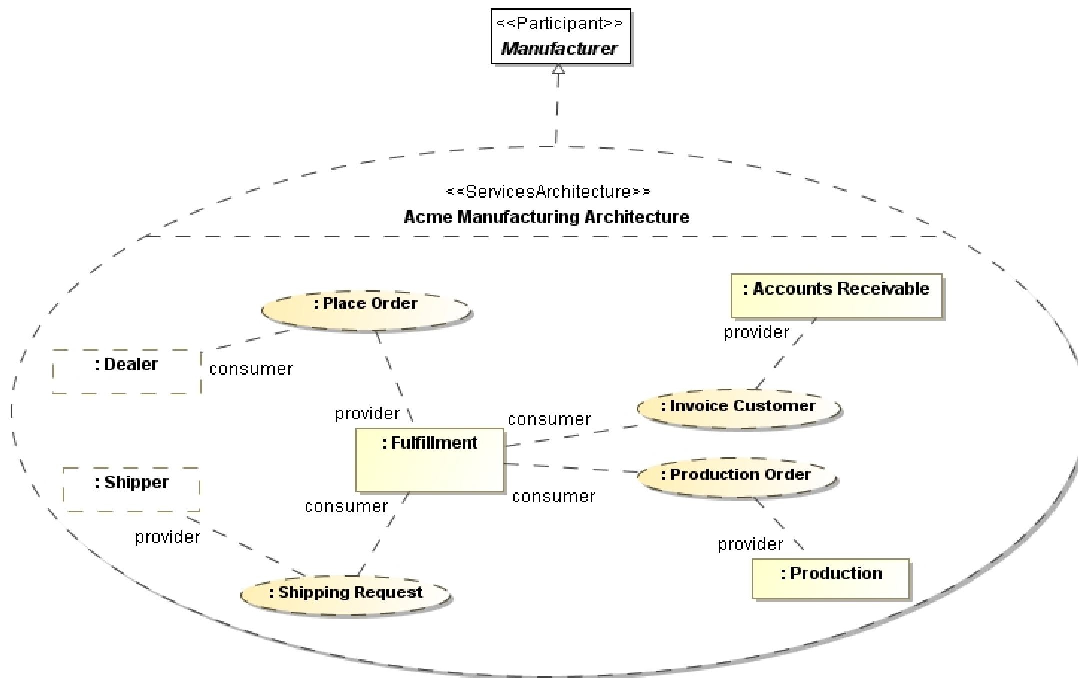


Figure 6.8 - Example Services architecture for a participant

Figure 6.8 illustrates the participant services architecture for a particular “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.

This services architecture is the architecture for a specific manufacturer, realizing the requirements of manufacturer in general. Note that the roles “outside” of the manufacturer are indicated by the roles with dashed outlines (:Dealer and :Shipper) where as the roles played by entities within Acme Manufacturing are normal roles. The roles with dashed lines are “Shared aggregation” in UML. A component can then either realize and/or use this services architecture.

The net effect is that services architectures may be used to specify how system and systems of systems work “all the way down.” These architectures can then be realized by any number of technology components.

6.2.3 Service Contracts and Contract based SOA

A key part of a service is the ServiceContract (see Figure 6.9).

Figure 6.9 - Example ServiceContract

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 that 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 or all participating parties in the case of a multi-party 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.

Each role, or party involved in a ServiceContract is defined by an Interface or ServiceInterface that is the type of the role. A ServiceContract is a binding contract - binding on any participant that has a service port typed by a role in a service contract. It defines the relationships between a set of roles defined by Interfaces and/or ServiceInterfaces.

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 6.10). 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.

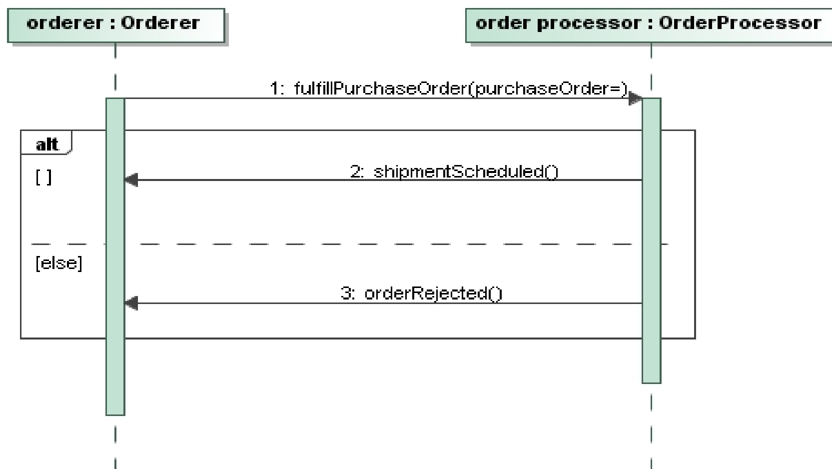


Figure 6.10 - Example choreography

The requirements for entities playing the roles in a ServiceContract are defined by Interfaces or 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 bi-directional interactions or communications between the participating roles - 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 port on a participant - possibly designated as a Service or Request. 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 services architecture or a composite component. Such participant can also use internal or external services, assemble other participants, business processes, and other forms of implementation. SoaML shows how the internal structure of a participant is described using other services. This is done by defining a ServicesArchitecture for participants in a more granular (larger scale) services architecture as is shown in Figure 6.7 and Figure 6.8.

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.

6.2.4 Example of Contract based SOA

The focus of a contract based SOA is the specification of the contract for services and how participants play their role in those service contracts as providers and consumers of the service. A service contract represents an agreement between parties for how they will carry out the service. This agreement may be established early when the services are defined or late when they are used. But, by the time the service actually happens, it is happening with respect to some service contract. Existing "bottom up" services are frequently adapted to these service contracts, which may be specified at an enterprise, community, or system level.

A service contract is represented as a UML Collaboration and defines specific roles each participant plays in the service contract. These roles have a name, an interface type that may be stereotyped as the "Provider" or "Consumer." The consumer is expected to start the service, calling on the provider to provide something of value to the consumer.

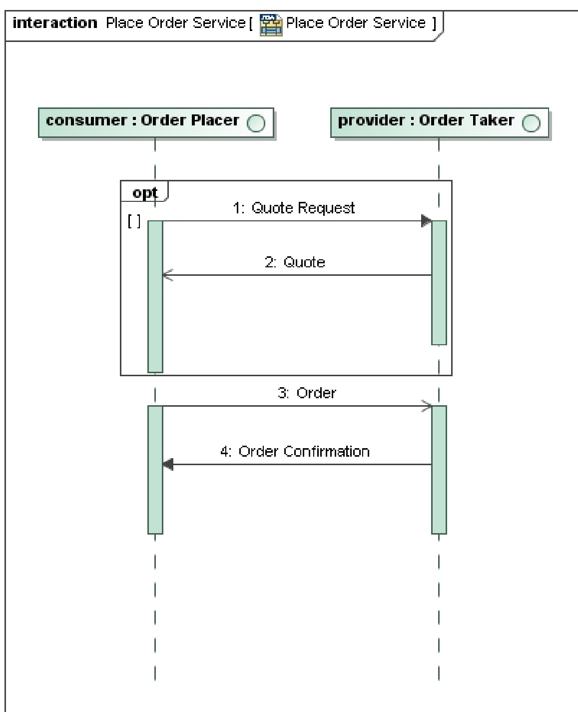


Figure 6.11 - Place Order Service Contract Choreography

Figure 6.11 illustrates the choreography of a service contract. The subject of this service contract is placing orders, which may optionally include a quotation. There are two primary interaction sets: 1) the quote request resulting in a quote, and 2) the order resulting in an order confirmation. In this case these interactions can be defined using signals, which makes this service contract asynchronous and document oriented, a mainstream SOA best practice.

Let's look at some of the parts of this service contract.

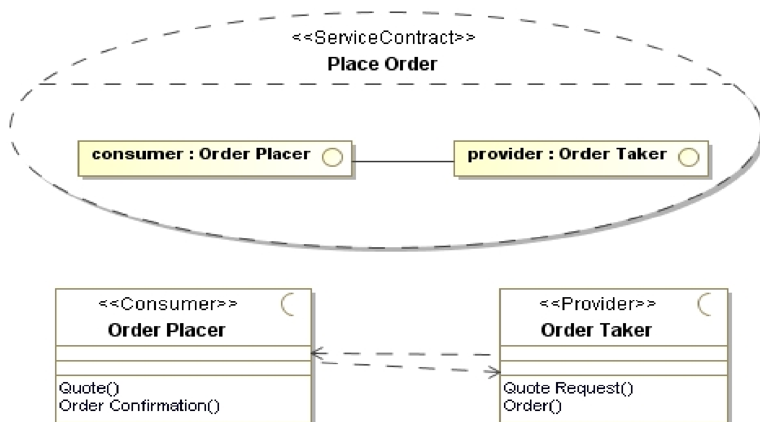


Figure 6.12 - Service Contract Collaboration and Interfaces

The interaction diagram in Figure 6.12 is part of the place order service contract collaboration. This collaboration binds the parts of the service contract together. The parts include the interaction diagram; describing the roles of the service, and the interfaces that define what operations and signal receptions may be received by the participants playing each role. Let's look at each part individually:

- **«ServiceContract»Place Order Service** – this is the service contract itself – defining the terms and conditions under which the service can be enacted and the results of the service. The service contract may be related to business goals or requirements. The service contract can also be used in services architectures to show how multiple services and participants work together for a business purpose.
- **provider : Order Taker** – this defines the role of the provider in the place order service. The provider is the participant that provides something of value to the consumer. The type of the provider role is "Order Taker," this is the interface that a provider will implement on a port to provide this service – note this also could be a «ServiceInterface».
- **consumer: Order Placer** – this is the role of the consumer in the place order service. The consumer is the participant that has some need and requests a service of a provider. The type of the consumer role is "Order Placer," this is the interface that a consumer will implement on a port to consume this service (note that in some cases this interface may be empty, indicating a one-way service).
- **«Provider» Order Taker** – this is the type of a place order service provider indicating all the operations and signals a providing participant may receive when enacting this service. Note that the order taker uses the order placer (the dashed line from the order taker to the order placer) – this shows that the order taker calls the order placer (using signals or operations). In any bi-directional service the provider will respond to the consumers requests using the order placer interface.
- **«Consumer» Order Placer –** this is the type of a place order service consumer. This indicates all of the operations and signals a consuming participant will receive when enacting the service with the provider. Note that the order taker uses the order taker (the dashed line from the order placer to the order taker) this indicates that the order placer calls the order taker. All consumer interfaces would use the provider interface. In simple unidirectional services, the consumer interface may be missing or empty.

6.2.4.1 Use of the service contract to define participants

The interfaces defined as part of a service contract represent how a participant must interact to participate in that service. A participant interacts via a UML “port.” A port is a point of interaction within UML and is used in SoaML as the interaction point for providing, consuming, or playing any other role in a service as defined by the service contract.

Let’s look at some participants.

Figure 6.13 - Participating in services

Note that both the dealer and manufacturer have a “Place order service” port, each typed by one of the interfaces in the place order service, above. Since the manufacturer’s place order service port has the “Order taker” type, it provides this interface. Since that interface requires the use of “Order Placer,” the manufacturer’s port requires that interface. These interfaces are described by the “Place order service” behavior and so must abide by that choreography when offering that service.

Likewise the “Dealer” has a “Place order service” port, but this time typed by the interface representing their role in the service - “Order Placer.” The dealer’s port then provides the order placer interface and requires the order taker. These ports have “conjugate” provided and required interfaces and are therefore compatible and these ports can be connected to enact the service.

Showing the port name, type, and interfaces is a bit of overkill, so we usually only show the port type on diagrams - but the full UML notation, above, should help to clarify how ports are used to provide and consume services. Note that these participants have other ports, showing that it is common for a participant to be the provider and consumer of many services.

The service contract is a binding contract with respect to the participants. By using the interfaces of a service contract (order placer and order taker in this example) the participants are bound to that contract when interacting via that port. The interfaces represent the type of each bound party. This is an extension to the UML concept of a collaboration that is not binding without an explicit collaboration use.

6.2.4.2 Use of “Service” and “Request” (Alternative)

An alternative way of using the same interfaces involves the use of “Service” and “Request” ports. When using this style of modeling the participant’s ports, only the Provider interface is used. The provider interface is used as the type of a “Service” and as the type of a Request. Request is a “conjugate” port, that is it inverts the required and provided interfaces. Use of Request and “Service” is useful when there is a simple one-way interface (that is the consumer does not have an interface) or for a binary service contract (as shown).

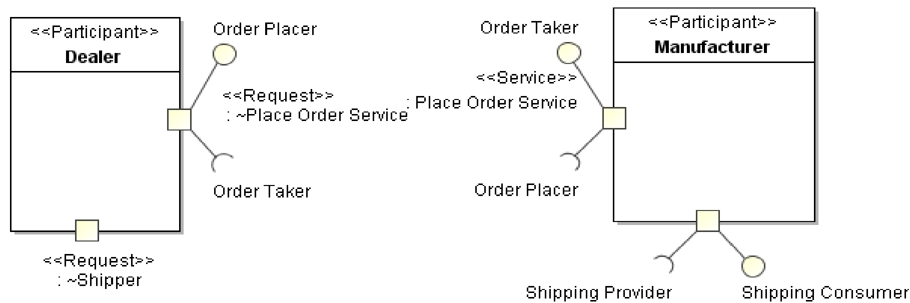


Figure 6.14 - Use of Request

Note that in the above diagram the type of the dealer's place order service port is "~Order Taker" - this is the order taker interface, the same as is used in the manufacturer's port. The tilde ("~") indicates that this is the conjugate type. Since the conjugated type has been used (as indicated by the Request stereotype) the interfaces that are provided and required by a port are exactly the same.

The use of the consumer's interface or Request yields exactly the same result - it is the modeler's option which way to use the service contract. For service contracts with more than two participants the Request method does not work as well.

6.2.4.3 Multi-Party Service Contracts

Multi-party service contracts are those with more than two participants - while this is a less common case, it does represent many business and technology interactions. Our example will use an escrow purchase. The service choreography could look like this:

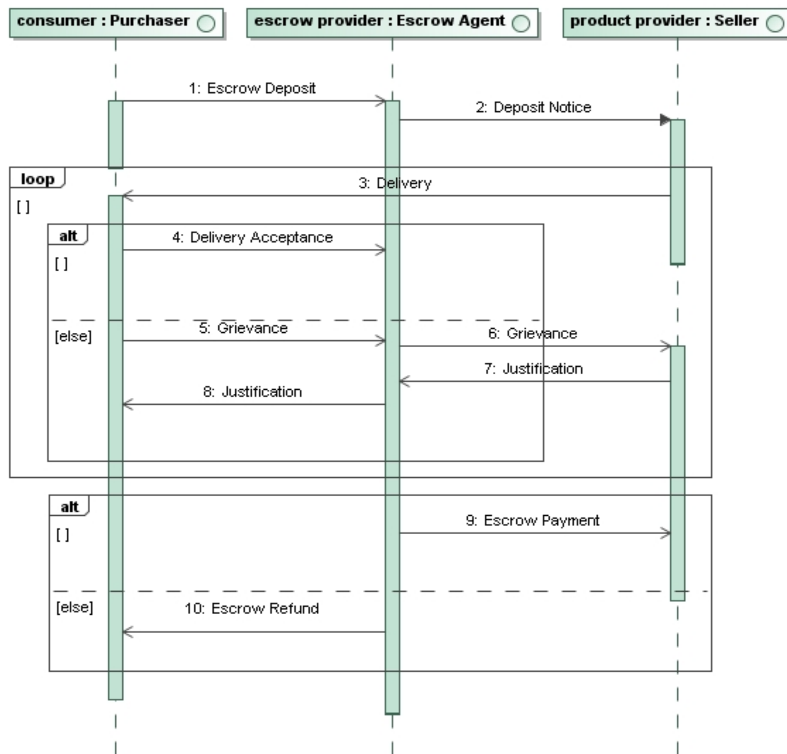


Figure 6.15 - Multi-party choreography

This shows a service starting with a deposit made by the purchaser to an escrow agent. At a later time a delivery is made and either accepted or a grievance is sent to the escrow agent who forwards it to the seller. The seller may file a justification. This process repeats until the escrow agent concludes the transaction and either makes the escrow payment to the seller (in the case where delivery was made) or refunds it to the buyer (if delivery was not made).

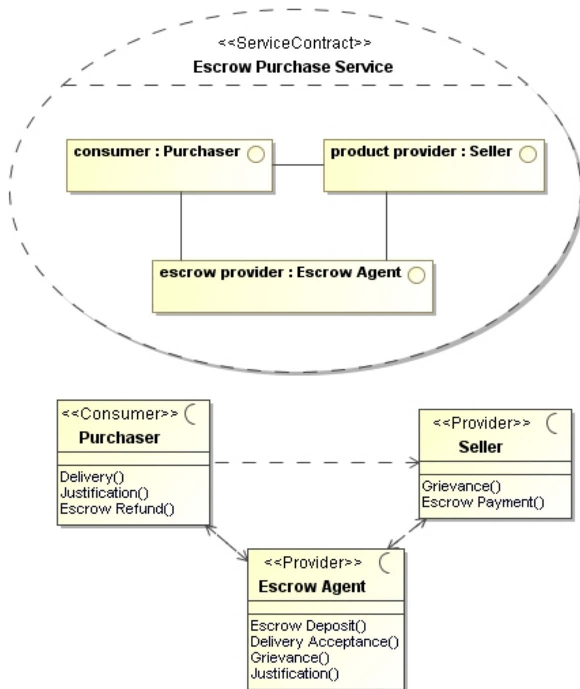
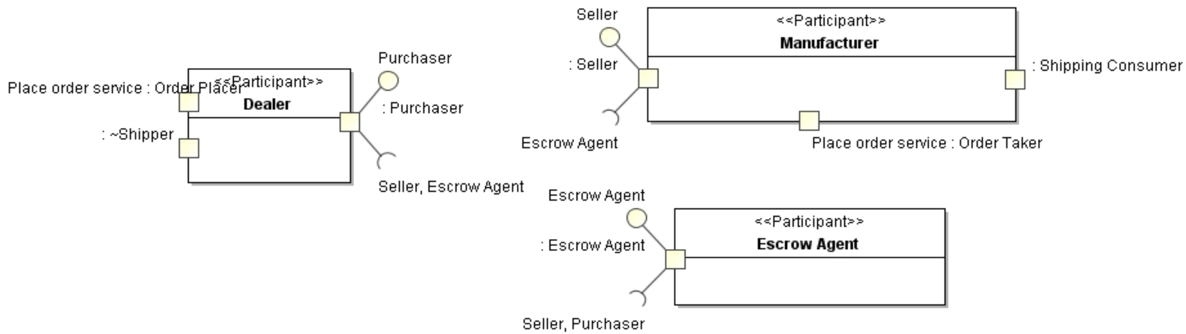


Figure 6.16 - Multi-party service contract and interfaces

This is a multi-party contract because the purchaser also interacts with the seller. This is a mediator pattern where except for delivery; the interaction between the purchaser and seller is mediated through the escrow agent.

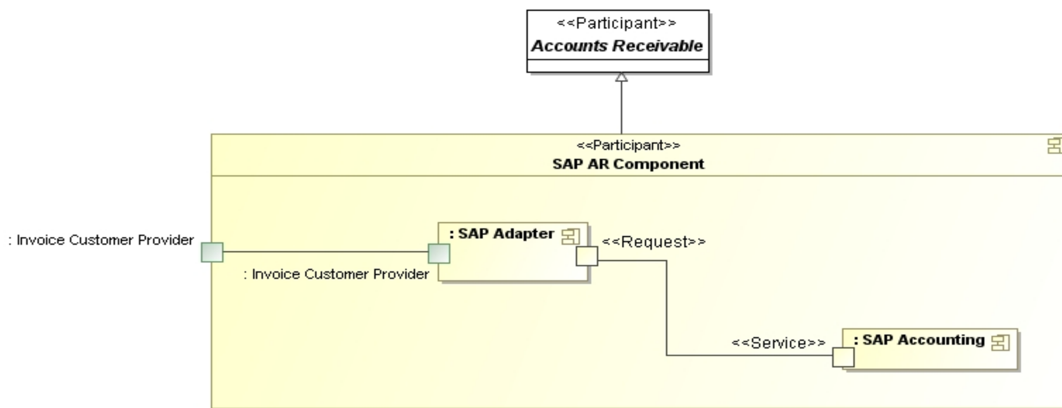
6.2.4.4 Participant ports in support of the multi-party service contract



As with the binary service contract, each participant provides their interface and uses the interfaces of each party they call, which is shown in terms of the interaction diagram as well as the dependencies between the interfaces. The choreography defines the rules for who calls who and when.

6.2.4.5 Adapting Existing Services

Top-down and bottom-up services are frequently joined with an adapter or wrapper pattern. These adapters may be created automatically or manually, depending on the supporting technology and the disparity between the services. Adaptation is used where a "bottom up" interface is provided but is overly coupled with the supporting technology. A top-down "enterprise service" is defined to be more general and less coupled. An adapter defines the connection between the two.



The figure above shows such an adapter pattern. The internal "SAP Accounting" module provides a (fictional) SAP interface where as the SAP AR Component complies with the enterprise service contract. The SAP adapter performs any required change in data, protocol, or process.

6.2.4.6 Defining a ServiceContract for a simple interface

For a simple interface to be used in a ServicesArchitecture it must be put in the context of a ServiceContract, as shown in Figure 6.17.

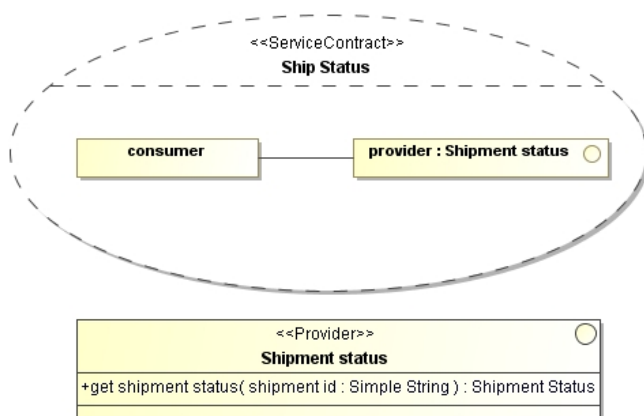


Figure 6.17 - Simple interface in service contract

Note that the simple interface is used as the provider, the consumer's role type may be empty and the consumer will use a Request. Adding a ServiceContract for a simple interface is then equivalent to the ServiceContract defined with one interface, above. The only difference being a top-down vs. bottom up approach to get to the same result. The consumer side of this service would use a "Request" port.

6.2.4.7 Showing how participants work together

While the dealer and manufacturer are compatible, nothing shows us how the set of participants work together in this SOA. How a set of participants work together, providing and using services, is shown in a ServicesArchitecture. Note that in this services architecture we have used the definition of both the participants and services to show how the parties collaborate.

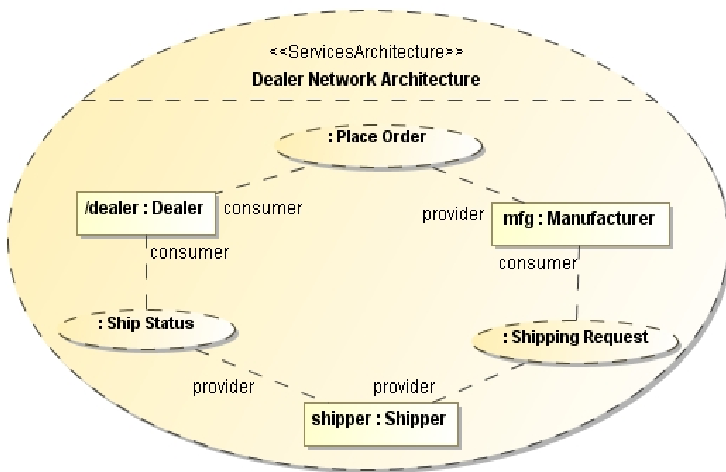
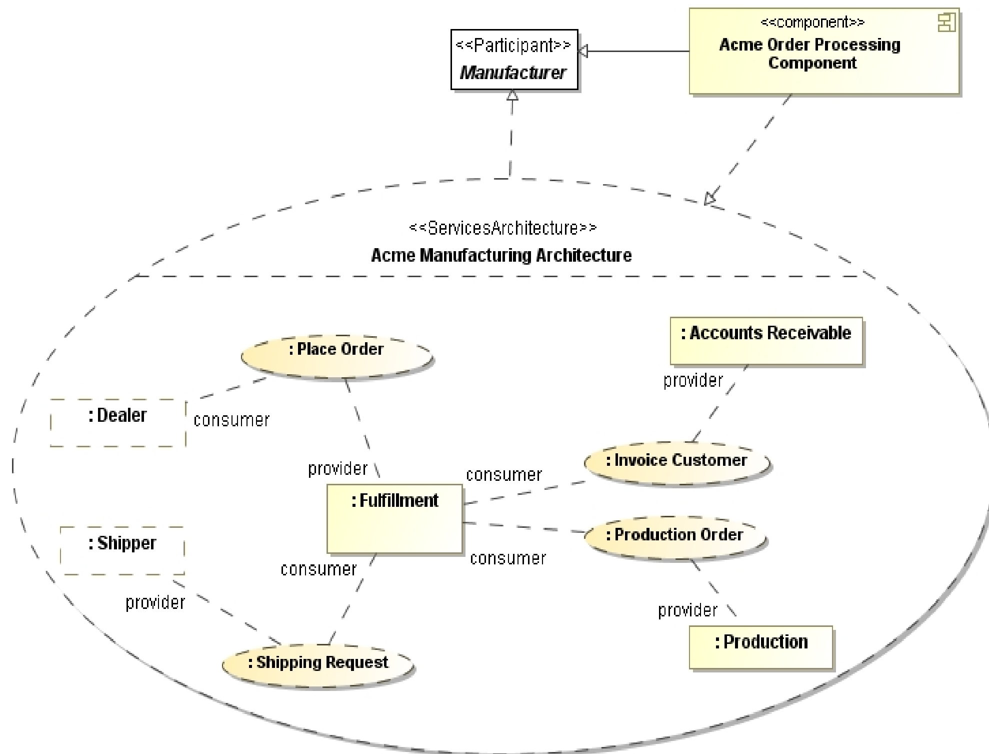


Figure 6.18 - Services Architecture

The above services architecture shows how the dealer, manufacturer, and shipper work together using the place order service, the shipping request service, and the ship status service. The participants defined are compatible with this architecture since they have ports that provide and use the corresponding interfaces.

6.2.4.8 Services Architecture for a Participant

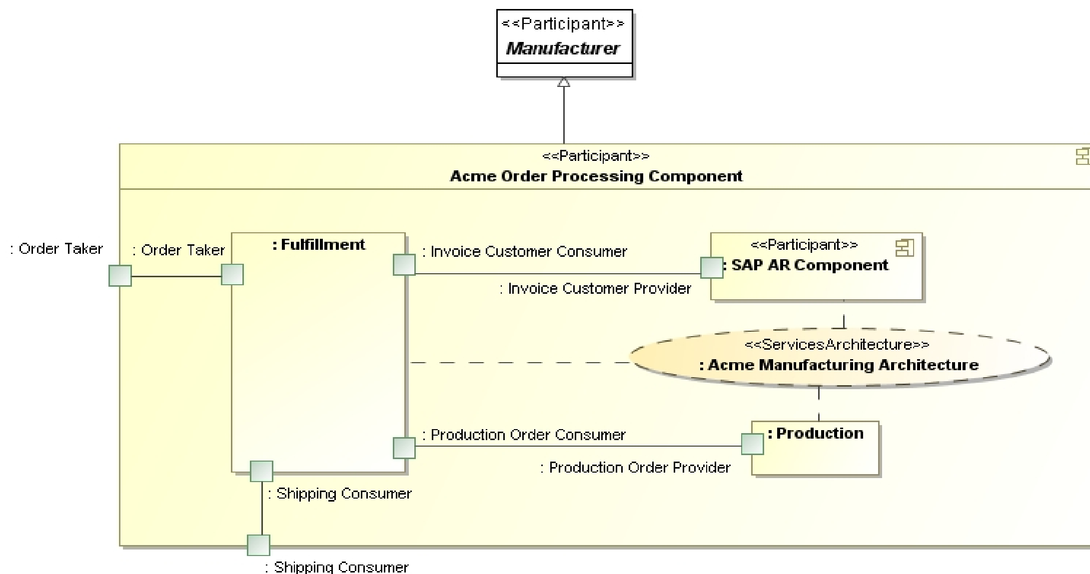
The dealer network services architecture defines how participants collaborate as a community. It is, of course, common for participants to collaborate within an organization. In this example we would like to see "inside" a particular manufacturer. Inside a Manufacturer, "Acme" in this example, business decisions have been made about how Acme is going to play its role in the dealer network. In UML terms Acme is realizing the manufacturer role in the dealer network. One way to express Acme's architecture is to provide a ServicesArchitecture that shows how Acme is going to be organized internally while satisfying the requirements of the dealer network architecture role - Manufacturer.



The diagram above shows the Acme Manufacturing Architecture. This realizes the Manufacturer participant and also serves as the specification for a technology component, the "Acme Order Processing Component."

The Acme Manufacturing Architecture shows how Acme is organized internally to provide services. This same architecture will frequently have a corresponding business process. Note that within Acme we have defined additional participant roles "Fulfillment," "Accounts Receivable," and "Production." Services are then used to connect these new roles with the existing, external roles "Dealer" and "Shipper." Dealer and Shipper use the "shared aggregation" feature of UML to show that these roles are external to Acme while the other roles are internal. The definition of this architecture will then define the participant component types for fulfillment, accounts receivable, and production.

Once the components are defined a composite structure is used to show how implementations of these components (such as SAP….) form a composite service oriented application.



The diagram above shows a composite application component for Acme. Note that component implementations (like the SAP AR Component) can be substituted for the more abstract participants. This component implements the Acme services architecture.

6.2.5 Capability

ServiceArchitectures and ServiceContracts 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, even the abstract Participants may not yet be 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.

6.2.5.1 Capabilities represent an abstraction of the ability to affect change

Capabilities 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 6.19 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 catalogs in order to communicate the needs and capabilities of a service area, whether that is 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 on 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.

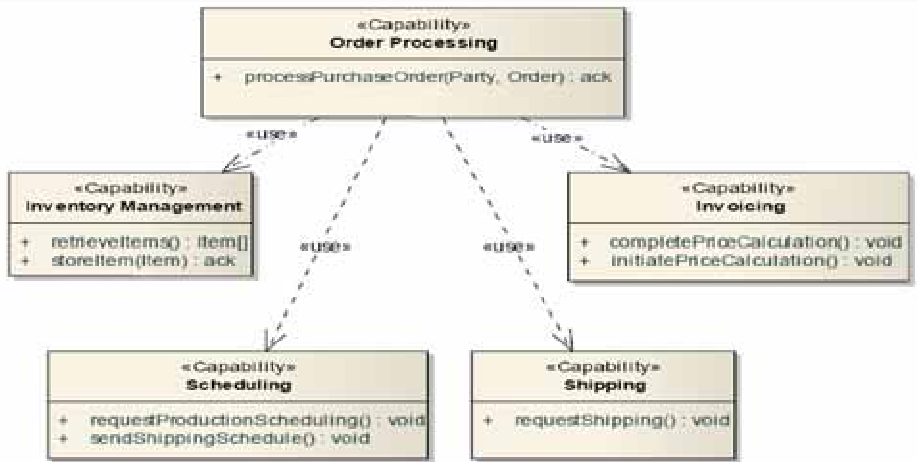


Figure 6.19 - Service Capabilities needed for processing purchase orders

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 6.20 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.

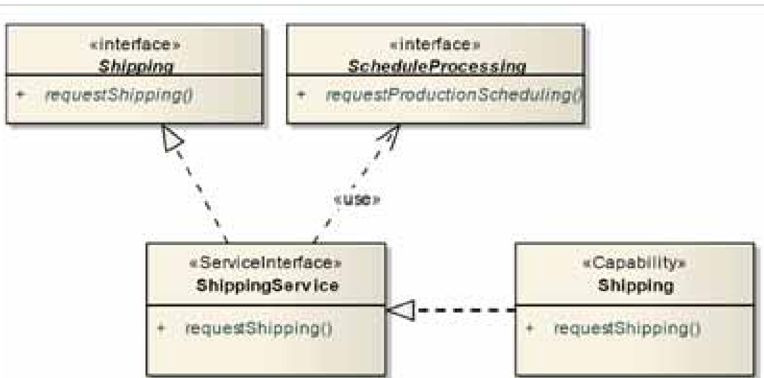


Figure 6.20 - ServiceInterface realized by a Capability

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 Service on the Participant as shown in Figure 6.21.

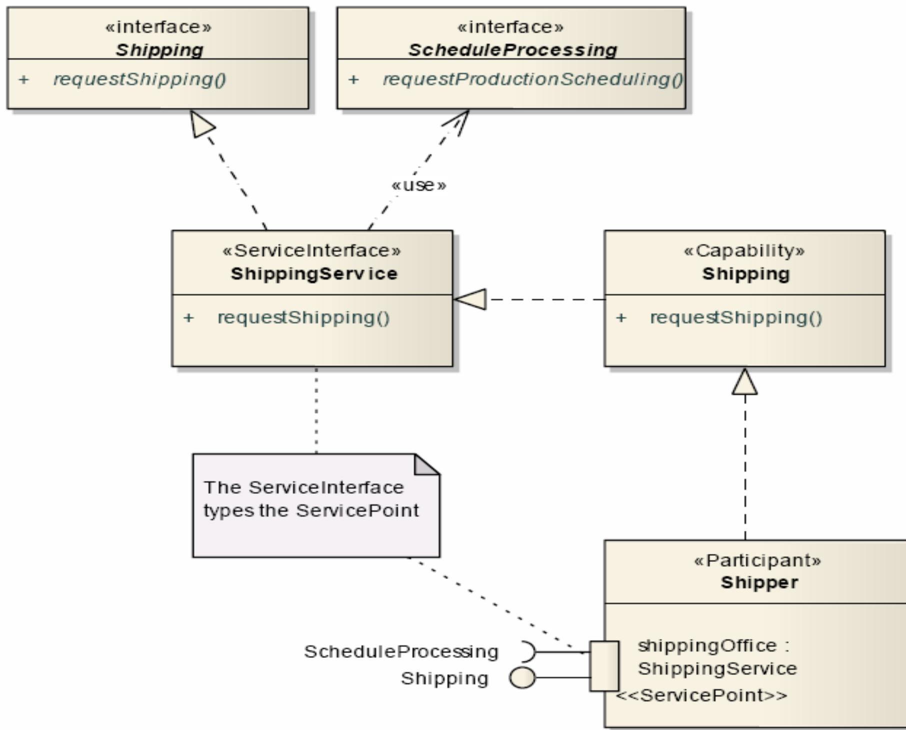


Figure 6.21 - The Shipper Participant realizes the Shipping Capability

Capabilities may also be used to specify the parts of Participants, the capabilities the participant has to actually provide its services. Figure 6.22 shows the Productions Participant with two parts typed by Capabilities. The productionOffice Service 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.

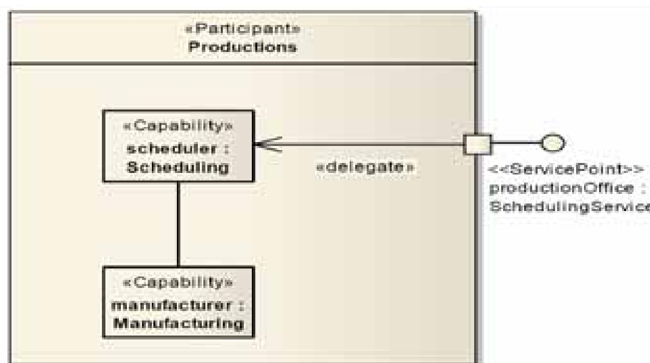


Figure 6.22 - Productions Participant with two parts specified by Capabilities

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 means of identifying candidate services or a more general notion of “providing access” to a general capability of a Participant. Figure 6.23 provides an example of such a situation.

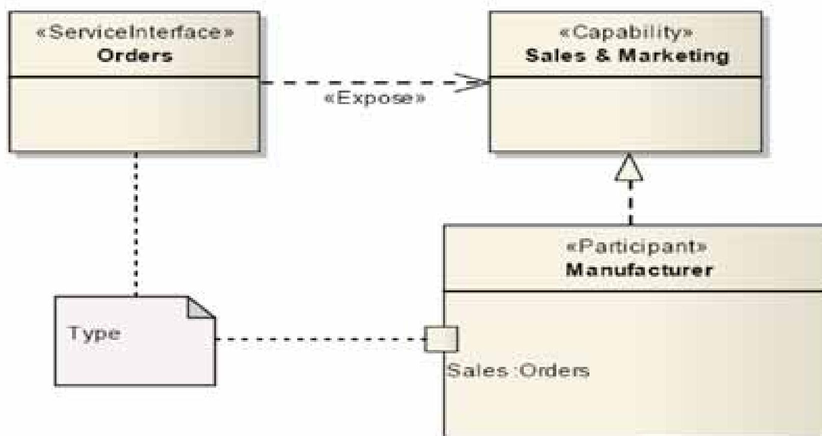


Figure 6.23 - The Orders ServiceInterface exposing the Sales and Marketing Capability

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.

6.2.6 Business Motivation

Services are intended to facilitate the agile development of business relevant solutions. To do so, services provide functional capabilities that when implemented and used provide some real-world effect that has value to potential producers and consumers. Business requirements can be captured using the OMG Business Motivation Model (BMM). BMM can be used to capture the influencers that motivate the business to change, the desired ends it wishes to achieve, and their supporting means. The ends may be defined by a business vision amplified by a set of goals and objectives representing some desired result. The means are the courses of action that when carried out support achievement of the ends producing the desired result as measured and verified by some assessment of the potential impact on the business. Courses of action may represent or be carried out by interactions between consumers with needs and providers with capabilities.

Any UML BehavioredClassifier (including a ServicesContract for example) may realize a BMM MotivationElement through a MotivationRealization. This allows services models to be connected to the business motivation and strategy linking the services to the things that make them business relevant.

However, the business concerns that drive ends and means often do not address the concerns necessary to realize courses of action through specific automated solutions in some execution environment. For example, business requirements may be fulfilled and automated by an IT solution based on a SOA. Such a solution will need to address many IT concerns such as distribution, performance, security, data integrity, concurrency management, availability, etc. It may be desirable to keep the business and IT concerns separate in order to facilitate agile business solutions while at the same time providing a means of linking solutions to the business requirements they fulfill, and verifying they do so with acceptable qualities of service and according to business policies and rules. Capabilities, ServiceContracts, and ServicesArchitectures provide a means of bridging between business concerns and SOA solutions by tying the business requirements realized by the contracts to the services and service participants that fulfill the contracts.

There are other ways of capturing requirements including use cases. A ServiceContract, which is also a classifier, can realize UseCases. In this case, the actors in the use case may also be used to type roles in the service contract, or the actors may realize the same Interfaces used to type the roles.

6.3 The SoaML Profile of UML

The stereotypes, below, define how to use SoaML in UML based on the set of stereotypes defined in the SoaML profile.

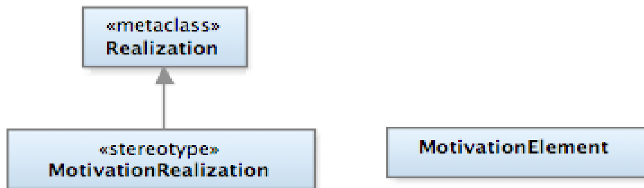


Figure 6.24 - BMM Integration

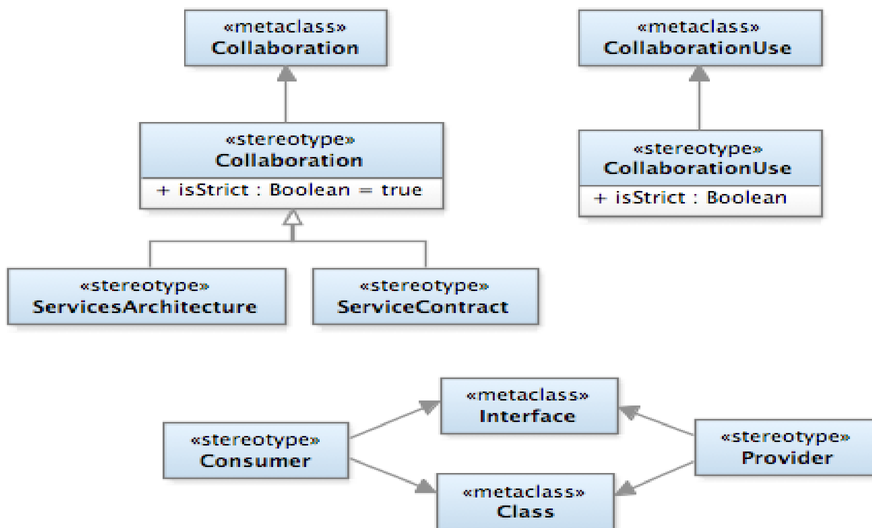


Figure 6.25 - Contracts

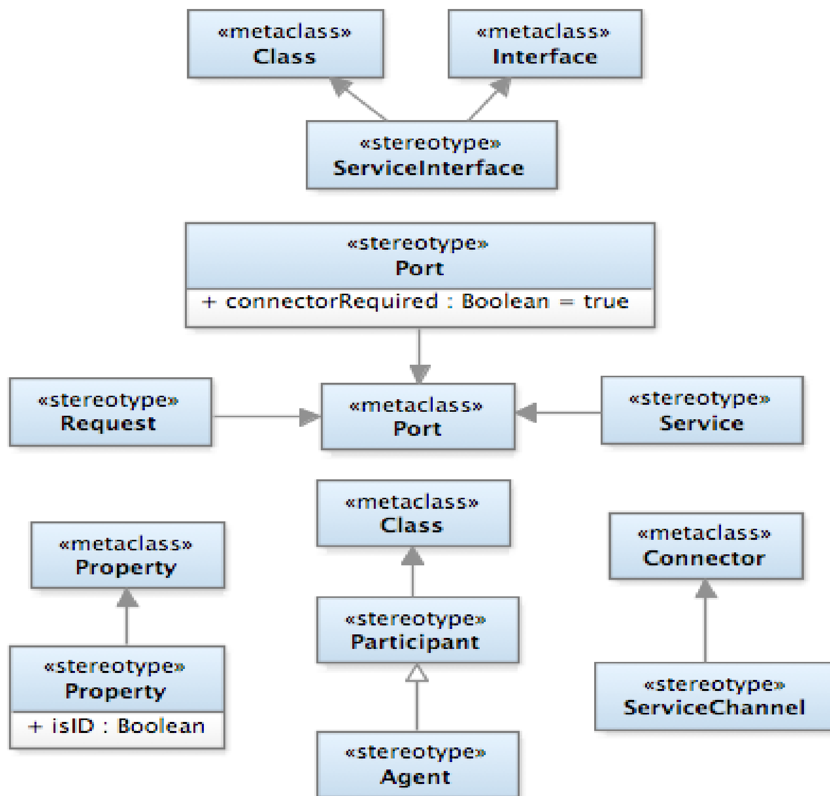


Figure 6.26 - Services

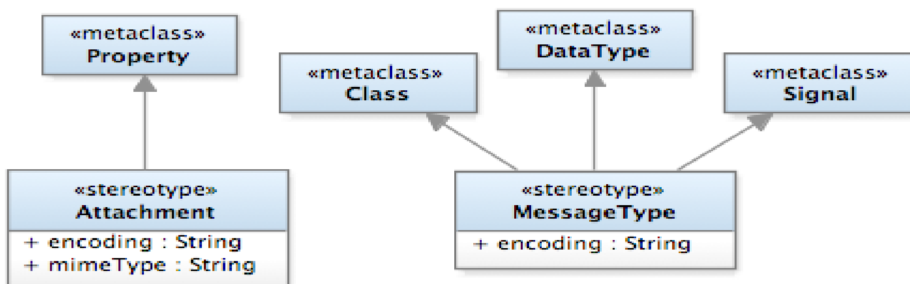


Figure 6.27 - Service Data

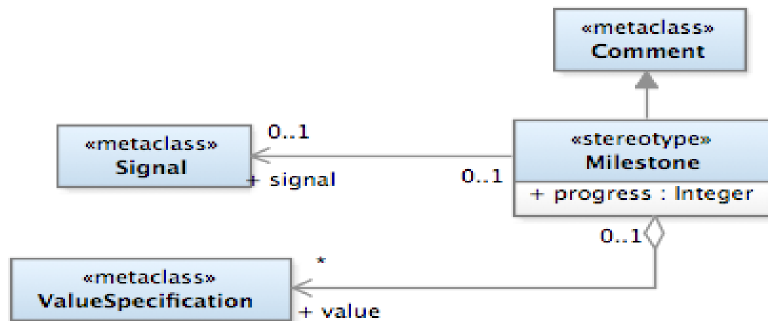


Figure 6.28 - Milestones

Figure 6.29 - Capabilities

6.4 Stereotype Descriptions

6.4.1 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. [1]

1. 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.

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.
- **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 ownedBehaviors that can be represented by UML2 Behaviors 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:

1. Autonomous – capable of 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.

2. 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 Woodridge-Jennings definition of agency, this property is referred to as *social ability*.

3. 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 Woodridge-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 a 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 life-cycle, 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 or the Agent icon decoration in the classifier name compartment.

Figure 6.30 - Agent notation

Additions to UML2

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

6.4.2 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 message that 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.
- mimeType: String [0..1]
  Denotes the iana MIME media type for the Attachment See: http://www.iana.org/assignments/media-types/.

Associations

No additional associations

Constraints

No additional constraints

Semantics

In a 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 6.31 shows an InvoiceContent Attachment to the Invoice MessageType. This attachment contains the detailed information about the Invoice.

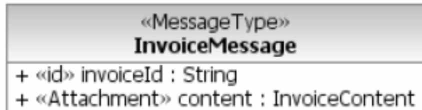


Figure 6.31 - The InvoiceContent Attachment

Additions to UML2

Extends UML2 to distinguish message attachments from other message proprieties.

6.4.3 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 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 catalogs 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. Figure 6.19 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 or Capability icon decoration: .

Examples

For examples of Capability, see from Figure 6.19 to Figure 6.23.

Additions to UML2

Capability is a new stereotype used to describe service capabilities.

6.4.4 Consumer

Consumer models the type of a service consumer. A consumer is then used as the type of a role in a service contract and the type of a port on a participant.

Extends Metaclass

- Interface (in the case of a non composite service contract)
- Class (in the case of a composite service contract)

Description

A «Consumer» models the interface provided by the consumer of a service. The consumer of the service receives the results of the service interaction. The consumer will normally be the one that initiates the service interaction. Consumer interfaces are used as the type of a «ServiceContract» and are bound by the terms and conditions of that service contract.

The «Consumer» is intended to be used as the port type of a participant that uses a service.

Attributes

No additional attributes

Associations

No additional Associations

Constraints

The «Consumer» is bound by the constraints and behavior of the ServiceContract of which it is a type.

Semantics

The concept of a provider and a consumer is central to the concept of a service oriented architecture. The consumer requests a service of the provider who then uses their capabilities to fulfill the service request and ultimately deliver value to the consumer. The interaction between the provider and consumer is governed by a «ServiceContract» where both parties are (directly or indirectly) bound by that contract.

The consumer interface and therefore the consumer role combine to fully define a service from the perspective of the consumer.

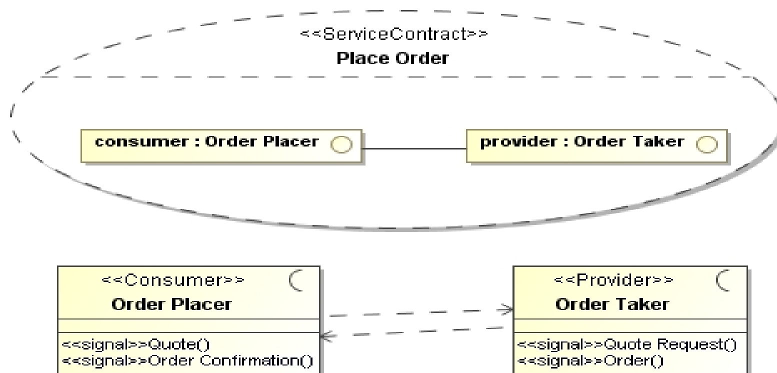
The consumer interface represents the operations and signals (if any) that the consumer will receive during the service interaction.

The Consumer will also have a uses dependency on the provider interface, representing the fact that the consumer must call the provider.

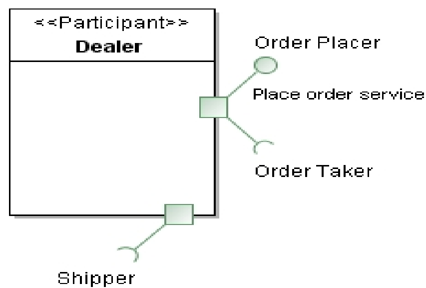
Notation

A Consumer is denoted using a Class or Interface with the «Consumer» stereotype.

Examples



The above diagram shows a consumer interface used as the type of a consumer role in a service contract. This consumer interface is then the type of a port on a participant that consumes this service.



The above diagram shows the consumer as the type of a participant's port where the service is consumed.

6.4.5 Collaboration

Collaboration is extended to indicate whether the role to part bindings of CollaborationUses typed by a Collaboration are strictly enforced or not.

Extends Metaclass

- Collaboration

Description

A Collaboration, ServiceContract, or ServicesArchitecture represents a pattern of interaction between roles. This interaction may be informal and loosely defined as in a requirements sketch. Or it may represent formal agreements or requirements that must be fulfilled exactly. A Collaboration's isStrict property establishes the default value of the isStrict property for any CollaborationUse typed by the Collaboration.

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 = true
  Indicates whether this Collaboration is intended to represent a strict pattern of interaction. Establishes the default value for any CollaborationUse typed by this Collaboration.

Associations

No new associations

Constraints

No new constraints

Semantics

A Collaboration is a description of a pattern of interaction between roles responsible for providing operations whose use can be described by ownedBehaviors of the Collaboration. It is a description of possible structure and behavior that may be played by other parts of the model.

A Collaboration may have isStrict=true indicating the collaboration represents a formal interaction between its roles that all parts playing those roles are intended to follow. If isStrict=false, then the collaboration represents an informal pattern of interaction that may be used to document the intended interaction between parts without specifically requiring parts bound to roles in CollaborationUses typed by the collaboration to be compatible. The isStrict property of a Collaboration establishes the default value for the isStrict property of all CollaborationUses typed by the Collaboration. A CollaborationUse may have this value changed to address particular situations.

Notation

No new notation

6.4.6 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 that 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 though the type may not be compatible. The default value is the value of the isStrict property of Collaboration used as the type of the CollaborationUse.

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:

1. The role and part have the same type.
2. The part has a type that specializes the type of the role.
3. The part has a type that realizes the type of the role.
4. 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 the “ServiceContracts” section 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 6.32 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.

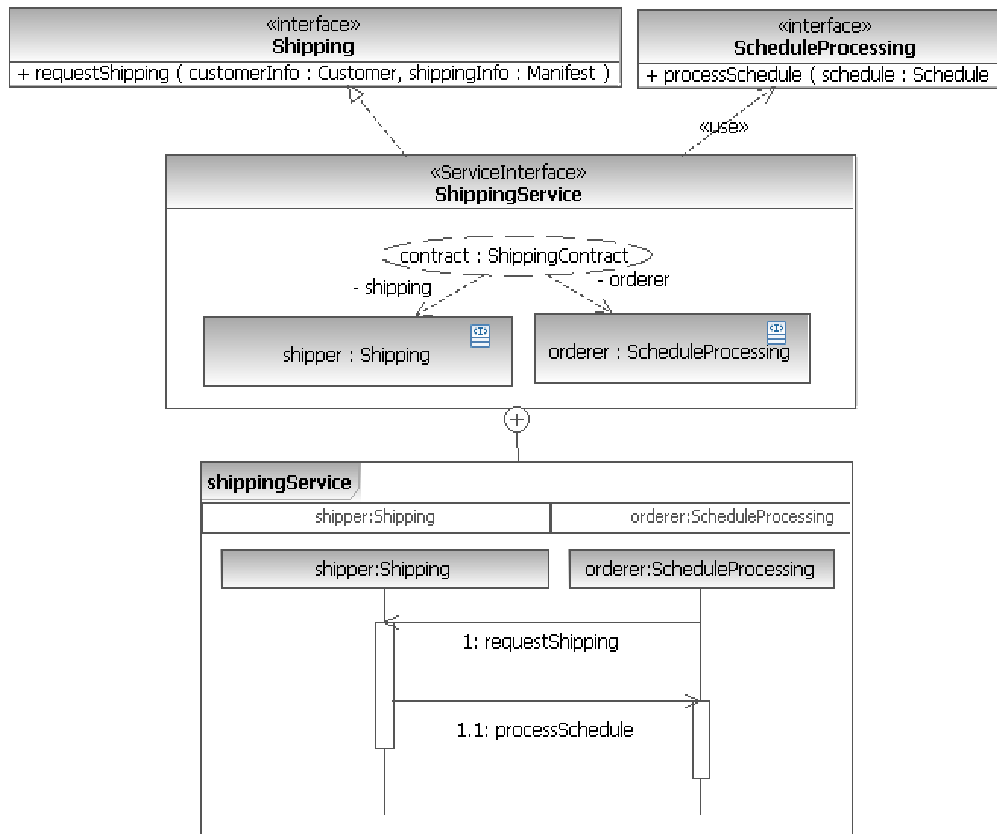


Figure 6.32 - Fulfilling the ShippingContract ServiceContract

Figure 6.33 shows a Participant that assembles and connects a number of other Participants in order to adhere to the Manufacturer Architecture 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.

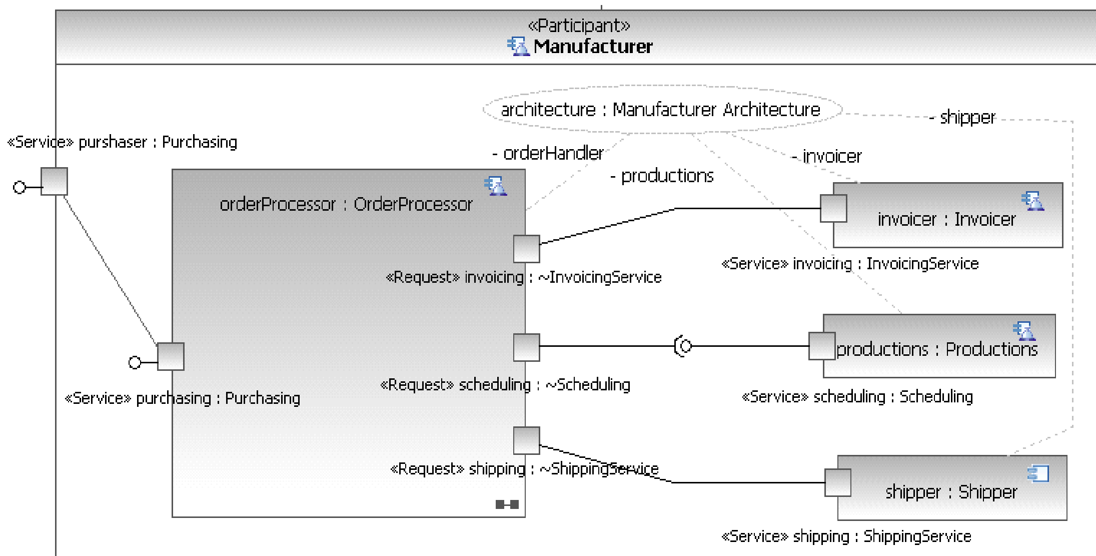


Figure 6.33 - Fulfilling the Process Purchase Order Contract

The orderProcessor part is bound to the orderHandler 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 invoicer 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.

6.4.7 Expose

An Expose dependency is used to indicate a Capability exposed through a ServiceInterface. The source of the Expose is the ServiceInterface, the target is the exposed Capability.

Extends Metaclass

- Dependency

Description

The Expose dependency provides the ability to indicate what Capabilities that are required by or are provided by a participant should be exposed through a Service Interface.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A Capability represents something a Participant needs to have or be able to do in order to support a value proposition or achieve its goals, or something a Participant has that enables it to carry out its provided services. Capabilities may be used to identify services that are needed or to describe the operations that must be provided by one or more services. An Expose dependency is a relationship between a Service Interface and a Capability it exposes or that provides it. This means that the exposing Service Interface provides operations and information consistent with the capabilities it exposes. This is not the same as realization. The Service Interface is not required to have the same operations and properties as the capabilities it exposes. It is possible that services supported by capabilities could be refactored to address commonality and variability across a number of exposed capabilities, or to address other SOA concerns not accounted for in the capabilities.

Alternatively, Capabilities may realize ServiceInterfaces using standard UML Realization. This approach is somewhat different in that it says that the Service Interface is a "specification" and the Capability "implements" this specification. As with the Expose dependency, the Capability is not required to have the same operations or properties as the Service Interface it realizes.

A Participant may have parts typed by Capabilities that indicate what the participant is able to do. Such a Participant may also Realize a set of Capabilities through its ownedBehaviors or through delegation to parts in its internal structure, or through delegation to requests for services from others. These capabilities may also be exposed by ServiceInterfaces used to specify the type of service ports through which the capabilities are accessed.

Notation

An Expose is denoted as a UML2 Dependency with the "Expose" stereotype.

Additions to UML2

Extends Dependency to formalize the notion that a Capability can be exposed by a ServiceInterface.

6.4.8 MessageType

The specification of information exchanged between service consumers and providers.

Extends Metaclass

- DataType
- Class
- Signal

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 to 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

[1] MessageType cannot contain ownedOperations.

[2] MessageType cannot contain ownedBehaviors.

[3] All ownedAttributes must be Public.

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 Service or Request. 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.

It is the intent of message type that it represents data values that can be sent between participants. Where message types contain classes as attributes or aggregated associations the message type will contain a “copy by value” of the public state of the those objects. Where those objects contain references to other objects those references will likewise be converted to value data types.

The way in which copy by value of an object is performed or how references are mapped to data types is platform technology dependent and not specified by SoaML.

It is the intent of message type that it represents data values that can be sent between participants. Where message types contain classes as attributes or aggregated associations the message type will contain a “copy by value” of the public state of the those objects. Where those objects contain references to other objects those references will likewise be converted to value data types.

The way in which copy by value of an object is performed or how references are mapped to data types is platform technology dependent and not specified by SoaML.

Notation

A MessageType is denoted as a UML2 DataType with the “MessageType” keyword.

Examples

Figure 6.34 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.

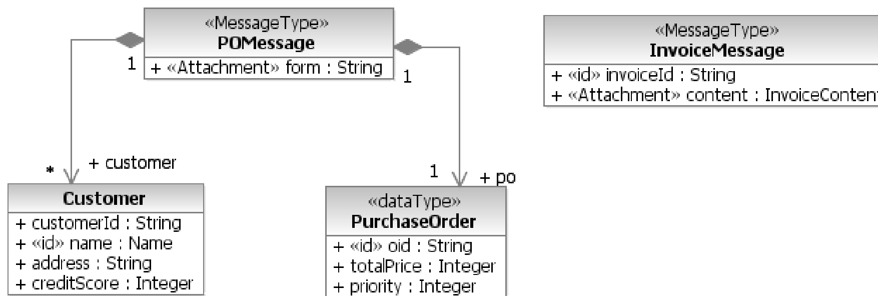


Figure 6.34 - MessageTypes in Purchase Order Processing

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 6.35 shows two examples of the Purchasing Interface and 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 that 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.

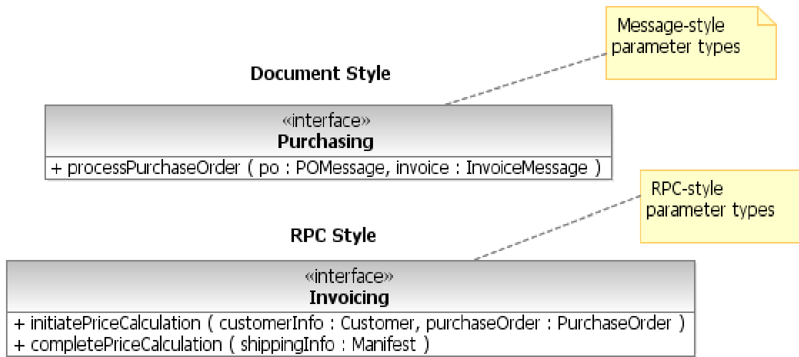


Figure 6.35 - Document and RPC Style service operation parameters

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.

6.4.9 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 [*]
  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 value is the signal name followed by the expression for the signal value in parenthesis.

Examples

Figure 6.36 - Milestones on Ordering Behavior

In Figure 6.36 we have taken the ordering behavior from Figure 6.10 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.

6.4.10 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 ports. These ports may use the “Service” and “Request” stereotypes that 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 concrete Participant may participate in and/or adhere to any number of services architectures. A composite structure is generally used to define the concrete sub-components of the participant.

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

[1] A Participant cannot realize or use Interfaces directly; it must do so through service ports, which may be Service or Request.

[2] Note that the technology implementation of a component implementing a participant is not bound by the above rule in the case of it's 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 ports. 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. Provided services may be implemented either through delegation to its parts representing capabilities or resources required to perform the service or other participants having the required capabilities and resources, through requests for services from others, through the methods of the service operations provided by its owned behaviors, or through actions that respond to received events. The implementation of the service must be consistent with the operations, protocols, and constraints specified by the ServiceInterface.

UML2 provides three possible ways a Participant may implement a service operation:

1. 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.

2. 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.

3. 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.

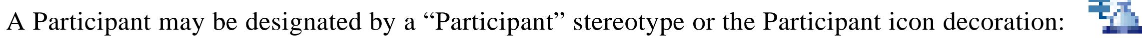
SoaML does not constrain how a particular participant implements its service operations. A single participant may mix delegation, method behaviors, and accept event actions to implement its services. A participant may also use different kinds of behavior to implement operations of the same service or 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 or the Participant icon decoration:



Specification Participants will have both the “Participant” and “specification” stereotypes.

Examples

Figure 6.37 shows an OrderProcessor Participant that provides the purchasing Service. This service provides the Purchasing Interface that 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.

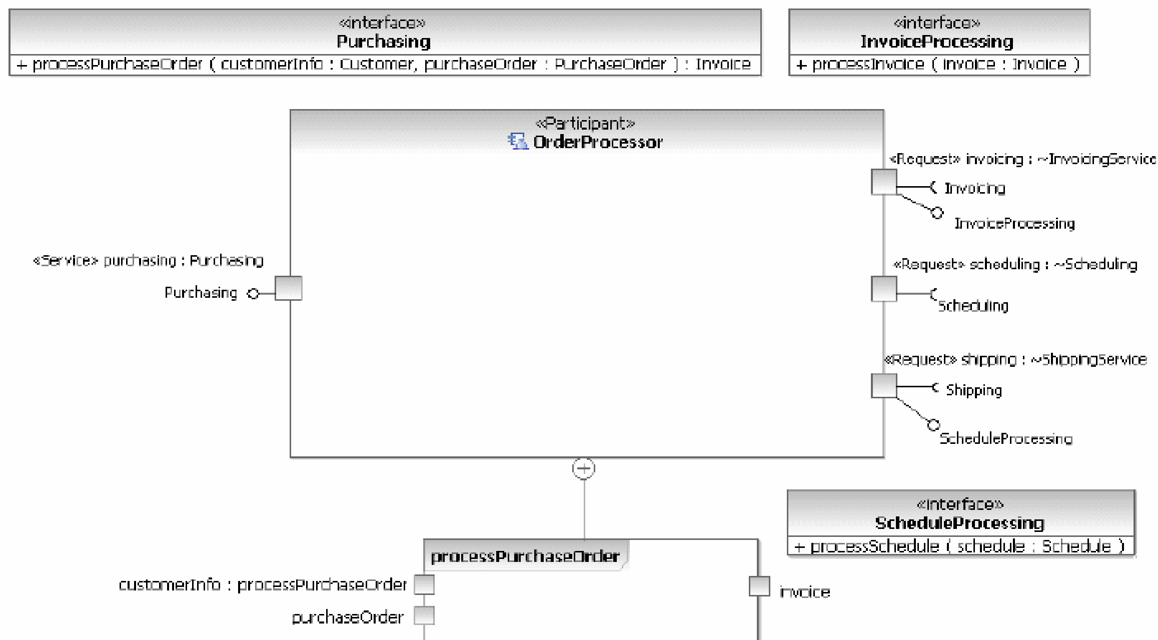


Figure 6.37 - The OrderProcessor Participant

Figure 6.38 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.

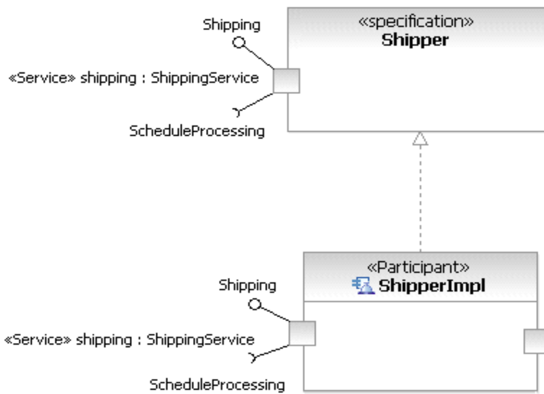


Figure 6.38 - The Shipper specification and the ShipperImpl realization Participants

Figure 6.39 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.

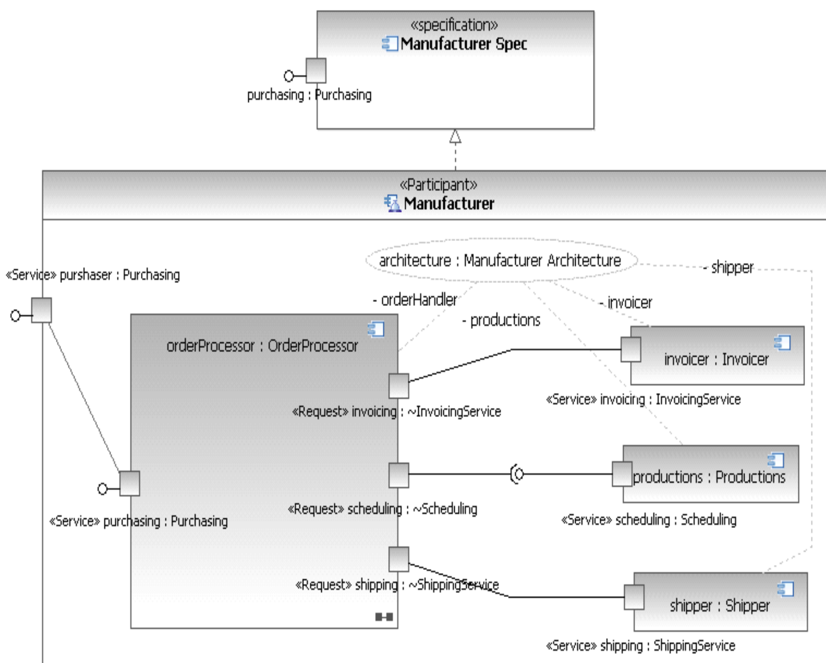


Figure 6.39 - The Manufacturer Participant

Additions to UML2

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

Request ports 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.

6.4.11 Port

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

Extends Metaclass

- Port

Description

Port is extended with a connectorRequired property to indicate whether a connector is required on this port, 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 Port 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 port that must be used, or a Request port 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 Port 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.

Port::isService is a convention supported by UML that recognizes upward, client-facing services a component might have as distinguished from downward services or requests that are used for implementation purposes and are not intended to be of interest to perspective clients. It is used to distinguish ports that the consumers are expected to be interested in from those that are public, but are mostly concerned with the implementation of the component through interaction with lower-level service providers. All these ports are either service or request ports, but isService is intended to distinguish those that would be involved in a client-facing value chain, and not something that is about the implementation of the participant or something provided for the detailed implementation of some other participant.

Notation

No additional Notation

Additions to UML2

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

6.4.12 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 that indicates isID=true.

Examples

Figure 6.34 shows an example of both data and message types with identifying properties.

Figure 6.38 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.

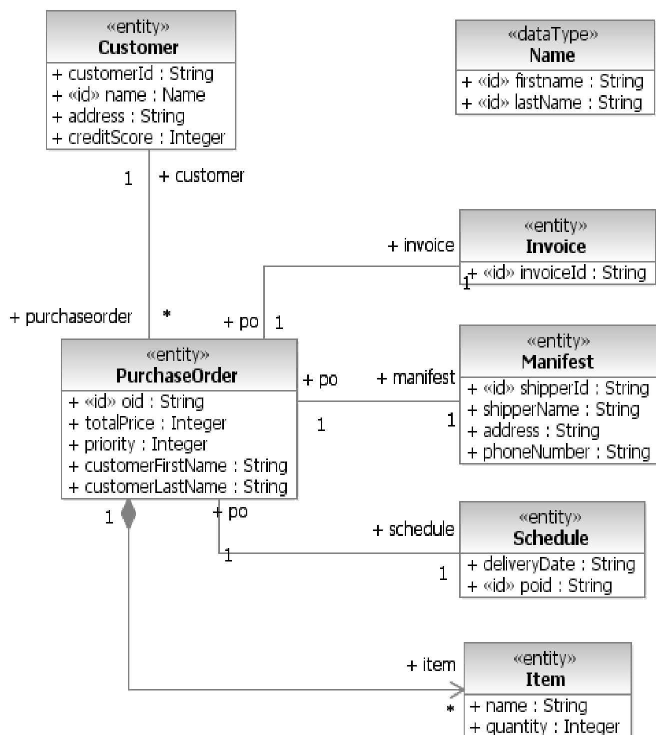


Figure 6.40 - Example entities from the CRM domain model

Additions to UML2

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

6.4.13 Provider

Provider models the type of a service provider in a consumer/provider relationship. A provider is then used as the type of a role in a service contract and the type of a port on a participant.

Extends Metaclass

- Interface (in the case of a non composite service contract)
- Class (in the case of a composite service contract)

Description

A "Provider" models the interface provided by the provider of a service. The provider of the service delivers the results of the service interaction. The provider will normally be the one that responds to the service interaction. Provider interfaces are used in as the type of a "ServiceContract" and are bound by the terms and conditions of that service contract.

The "Provider" interface is intended to be used as the port type of a participant that provides a service.

Attributes

No additional attributes

Associations

No additional Associations

Constraints

The "Provider" interface is bound by the constraints and behavior of the ServiceContract of which it is a type.

Semantics

The concept of a provider and a consumer is central to the concept of a service oriented architecture. The consumer requests a service of the provider who then uses their capabilities to fulfill the service request and ultimately deliver value to the consumer. The interaction between the provider and consumer is governed by a "ServiceContract" where both parties are (directly or indirectly) bound by that contract.

The provider interface and therefore the provider role combine to fully define a service from the perspective of the provider.

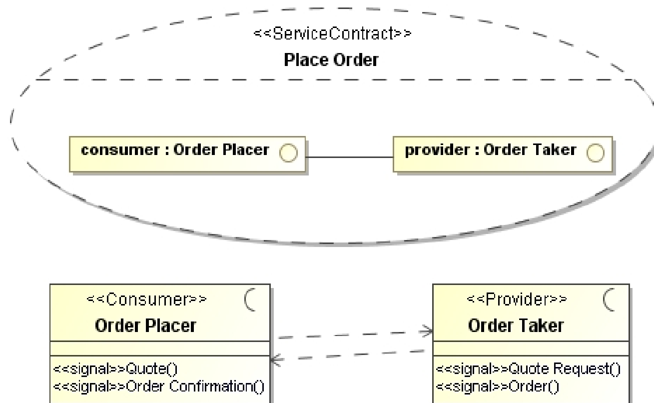
The provider interface represents the operations and signals that the provider will receive during the service interaction.

The Provider may also have a uses dependency on the consumer interface, representing the fact that the provider may call the consumer as part of a bi-directional interaction. These are also known as "callbacks" in many technologies.

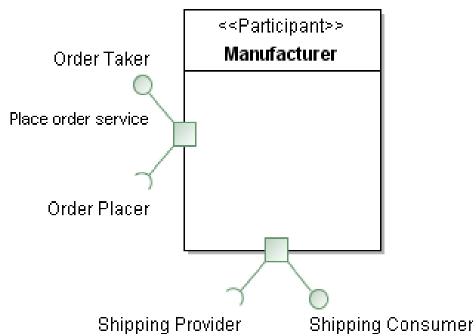
Notation

A Provider is a Class or Interface designated by the "Provider" stereotype.

Examples



The above diagram shows a provider interface used as the type of a provider role in a service contract. This consumer interface is then the type of a port on a participant that provides this service.



The above diagram shows the provider as the type of a participant's port where the service is provided.

6.4.14 Request

A Request represents a feature of a Participant that is the consumption of a service by one participant provided by others using well-defined terms, conditions and interfaces. A Request designates ports that define the connection point through which a Participant meets its needs through the consumption of services provided by others.

A request port is a "conjugate" port. This means that the provided and required interfaces of the port type are inverted; this creates a port that uses the port type rather than implementing the port type.

Extends Metaclass

- Port

Description

A Request extends Port to specify a feature of a Participant that represents a service the Participant needs and consumes from other participants. The request is defined by a ServiceInterface. It is used by the Participant either through delegation from its parts or through actions in its methods. The request may be connected to a business MotivationalElement to indicate the intended goals the Participant wishes to achieve. There may be constraints associated with the request that define its nonfunctional characteristics or expected qualities of service. This information may be used by potential providers to determine if their service meets the participant's needs.

A Request may include the specification of the value required from another, and the request to obtain value from another. A Request is the visible point at which consumer requests are connected to service providers, and through which they interact in order to produce some real world effect.

A Request 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 Request 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.

Request extends UML2 Port and changes how provided and required interfaces are interpreted by setting the ports isConjugated property to true. The capabilities consumed through the Request - its required interfaces - are derived from the interfaces realized by the service's ServiceInterface. The capabilities provided by a consumer 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 the 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 Request can connect to any Service as long as their types are compatible. Request 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

[1] The type of a Request must be a ServiceInterface or an Interface.

[2] The isConjugated property of a "Request" must be set to true.

Semantics

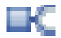
A Request represents an interaction point through which a consuming participant with needs interacts with a provider participant having compatible capabilities.

A Request 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 Request, 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 Request.

If the type of a “Request” is a ServiceInterface, then the Request’s provided Interfaces are the Interfaces used by the ServiceInterface while its required Interfaces are those realized by the ServiceInterface. If the type of a “Request” is a simple Interface, then the required interface is that Interface and the provided interfaces are those interfaces used be the simple interface, in any.

Notation

A Request may be designated by a Port with either a “RequestPoint” keyword and/or the Request icon decoration:



Examples

Figure 6.41 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.

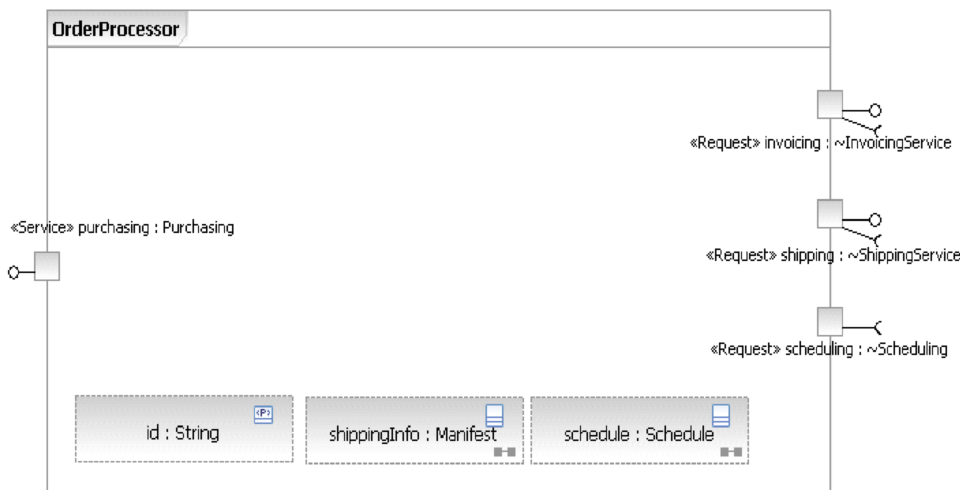


Figure 6.41 - Requests of the OrderProcessor Participant

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 “ServiceInterface” for details on these service interfaces. See “Service” for examples of Participants that provides services defined by these service interfaces.

Additions to UML2

None. “Request” uses the new isConjugated feature of UML Port.

6.4.15 ServiceChannel

A communication path between Services and Requests 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

[1] One end of a ServiceChannel must be a Request and the other a Service in an architecture.

[2] The Request and Service connected by a ServiceChannel must be compatible.

[3] 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 Request specifies a Participant's needs. A Service specifies a Participant's services offered. The type of a Request or Service 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 Port and Service Port 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 Request is compatible with, and may be connected to a Service through a ServiceChannel if:

1. The Request and Service have the same type, either an Interface or ServiceInterface.
2. The type of the Service is a specialization or realization of the type of the Request.
3. 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.
4. 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 6.42 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.

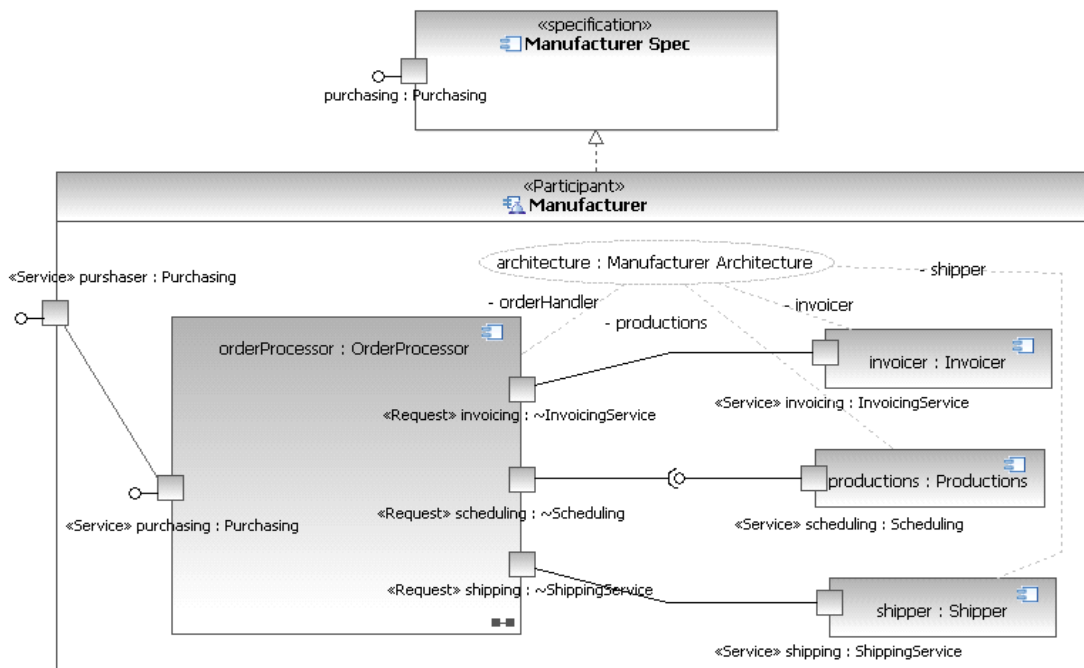


Figure 6.42 - The Manufacturer Participant

Additions to UML2

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

6.4.16 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 ServicesContract 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 or Class stereotyped as “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 Service or Request 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) - where 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, uses an Interface that in turn has operations that use 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:

1. As part of the Service Oriented Architecture (SOA), including services architectures, participant architectures, information models, and business processes.
2. 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.
3. Formalizing requirements and requirement fulfillment:
   - Without constraining the architecture for how those requirements might be realized.
   - Allowing for separation of concerns.
4. 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.
5. Defining and using patterns of services.
6. 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.
7. 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.
8. 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:

1. The role and part have the same type.
2. The part has a type that specializes the type of the role.
3. The part has a type that realizes the type of the role.

4. 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.
5. The type of each role in a service contract shall have a uses dependency to the type of all roles that role is connected to.

Semantics

Each ServiceContract role has a type that must be a UML Interface or (in the case of a composite service contract) a ServiceInteface or Class. At least one such interface must be stereotyped as a “Provider” and one as a “Consumer.” 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 an interface or class 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 6.43 is an example of a ServiceContract. The orderer and order processor participate in the contract.

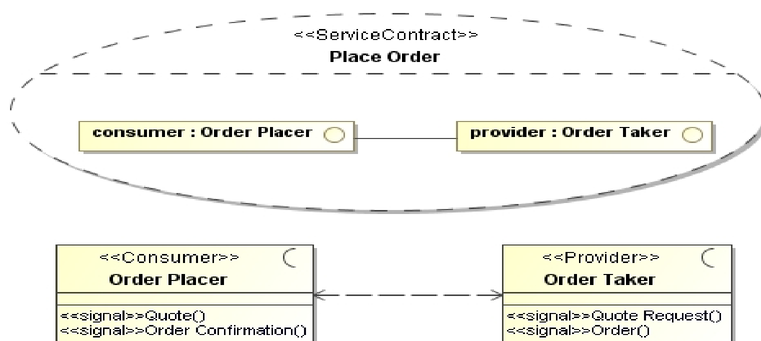


Figure 6.43 - The Ordering Service Contract

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.

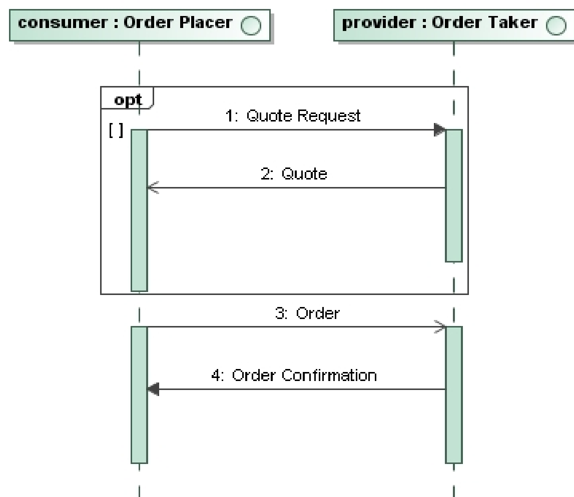


Figure 6.44 - Ordering Service communication protocol

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 shown below.

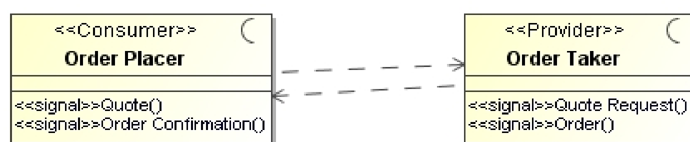


Figure 6.45 - Service interfaces that correspond to the above

Note that the above interfaces are the types of the roles in the ServiceContract shown in Figure 6.45.

The following example illustrates compound services.

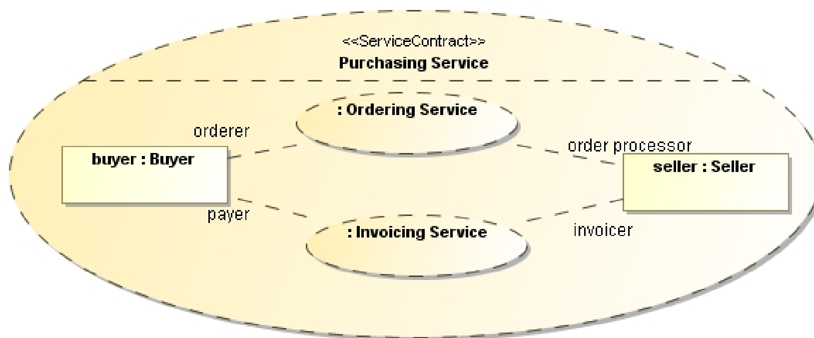


Figure 6.46 - Compound Services

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.

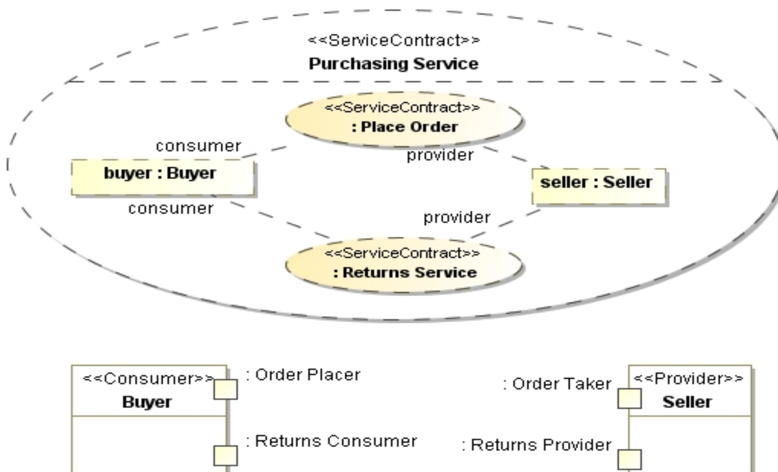


Figure 6.47 - Service Interfaces of a compound service

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» or using the ServiceContract icon decoration:

.

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.

6.4.17 ServiceInterface

Provides the definition of a service. Defines the specification of a service interaction as the type of a «Service» or «Request» port.

Extends Metaclass

- Class

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 Port. 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 port or Request port 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 self have service ports or request ports 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.

One or more ServiceInterfaces may also be combined in a ServiceContract which can further specify and constrain related services provided and consumed by Participants.

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

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

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 6.1.

Table 6.1 - Information in a ServiceInterface

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.	The provided Interfaces containing Operations modeling the capabilities. As in UML2, provided interfaces are designated using an InterfaceRealization between the ServiceInterface and other Interfaces.
Any service interaction consumers are expected to provide or consume in order to use or interact with a Service typed by this ServiceInterface.	Required Interfaces containing Operations modeling the needs. As in UML2, required interfaces are designated using a Usage between the ServiceInterface and other Interfaces.

Table 6.1 - Information in a ServiceInterface

The detailed specification of an interaction providing value as a service including: • Its name, often a verb phrase indicating what it does • Any required or optional service data inputs and outputs • Any preconditions consumers are expected to meet before using the capability • Any post conditions consumers participants can expect, and other providers must provide upon successful use of the service • 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	Each atomic interaction of a ServiceInterface is modeled as an Operation or event reception in its provided or required Interfaces. 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.
Any communication protocol or rules that determine when a consumer can use the capabilities or in what order	An ownedBehavior of the ServiceInterface. This behavior expresses the expected interaction between the consumers and providers of services typed by this ServiceInterface. The ownedBehavior could be any Behavior including Activity, Interaction, StateMachine, ProtocolStateMachine, or OpaqueBehavior.
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 ports or request ports on the service interface.

The semantics of a ServiceInterface are essentially the same as that for a UML2 Class which ServiceInterface extends. A ServiceInterface formalizes a pattern for using interfaces and behaviors, and the parts of a class to model interfaces to service protocol.

Participants specify their needs with Request ports and their capabilities with Service ports. Services and Requests, like any part, are described by their type which is either an Interface or a ServiceInterface. A request port may be connected to a compatible Service port 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 the receptions and operations it receives 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.

6.4.17.1 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 6.48 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 port or Request port is similar to using a WSDL PortType or Java interface as the type of an SCA component's service or reference.

Figure 6.48 - The StatusInterface as a simple service interface

Figure 6.49 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.

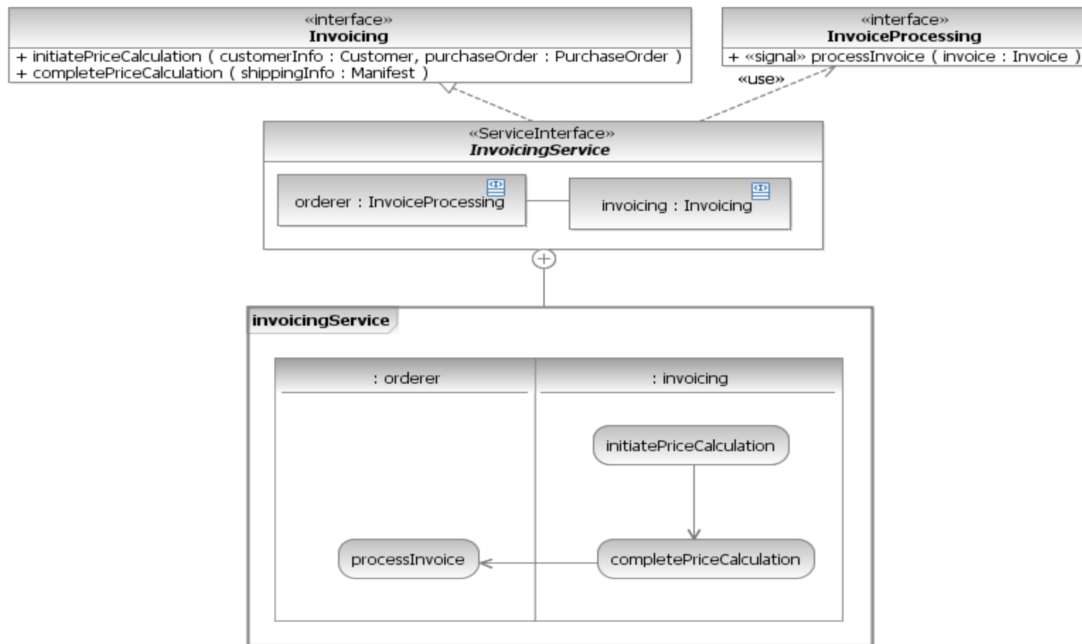


Figure 6.49 - The InvoicingService ServiceInterface

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 6.50 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.

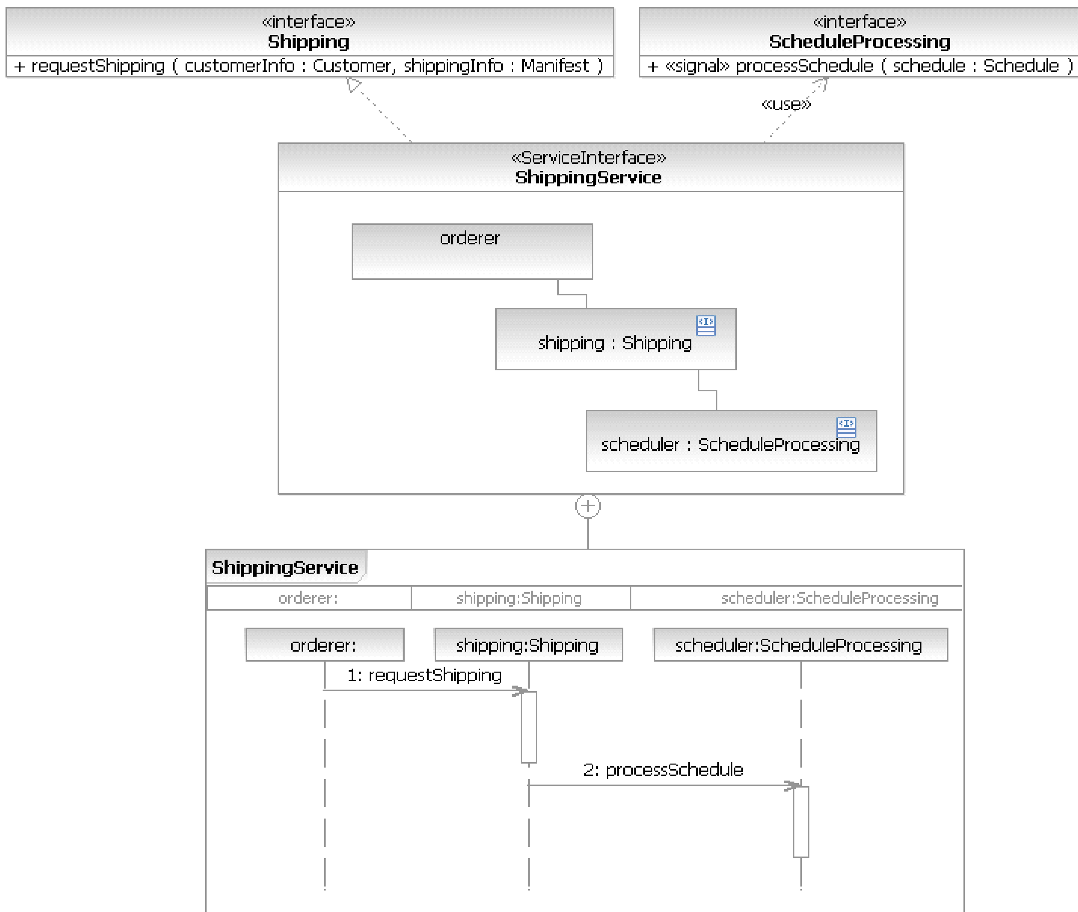


Figure 6.50 - Another version of the ShippingService supporting additional parties

Figure 6.51 shows an example where a Customer invokes the requestShipping operation of the shipping service, but a Scheduler 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 6.52 shows a different version of the OrderingSubsystem where a Customer both requests the shipping and processes the order. This ServiceChannel is also valid since this version of a Customer follows the complete protocol without depending on another part.

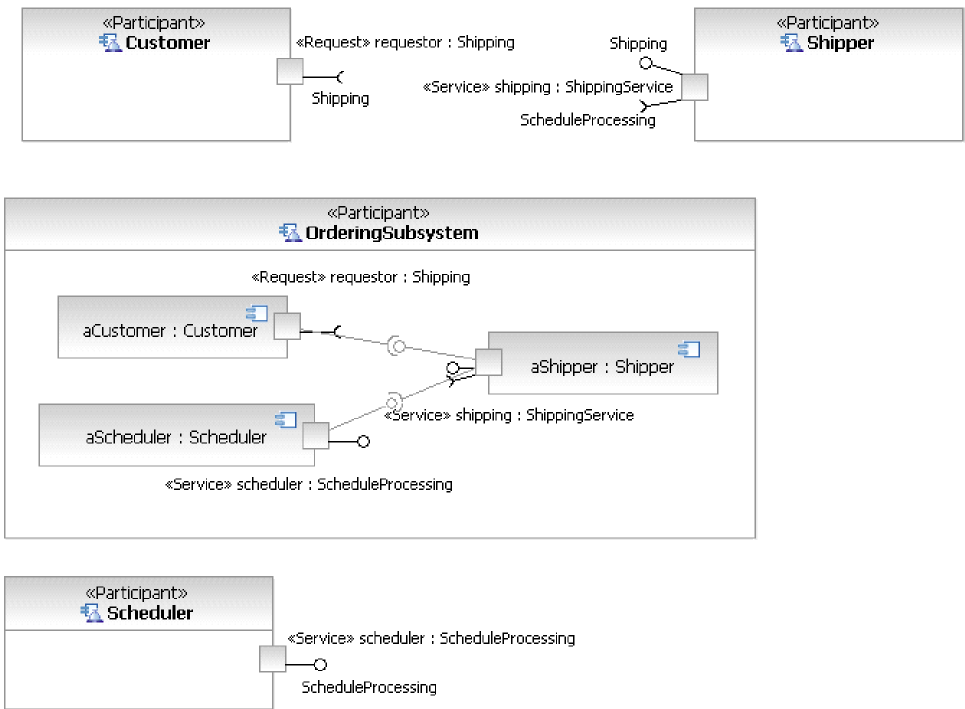


Figure 6.51 - Using the shipping Service with two Consumers

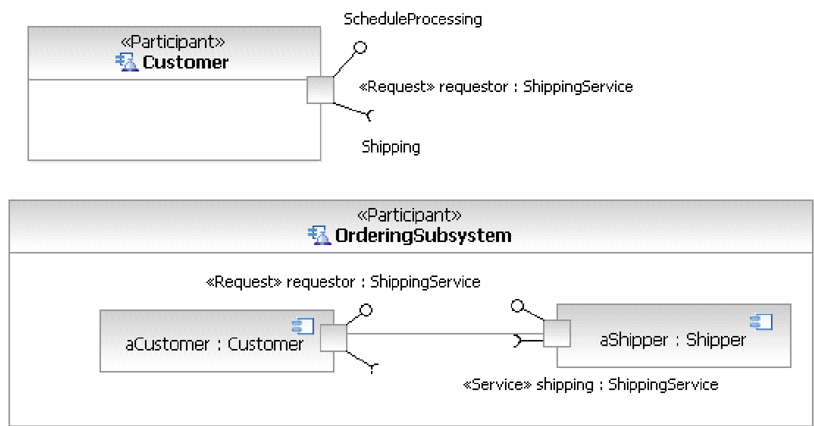


Figure 6.52 - Using the shipping Service with one Consumer

Additions to UML2

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

6.4.18 Service

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

Extends Metaclass

- Port

Description

A Service extends Port to specify a feature of a Participant that represents a service the Participant provides and offers for consumption by other participants. The service is defined by a ServiceInterface. It is implemented by the Participant either through delegation to its parts or through its methods. The service may be connected to a business MotivationalElement to indicate its intended value proposition. There may be constraints associated with the service that define its nonfunctional characteristics or warranted qualities of service. This information may be used by potential consumers to determine if the service meets their needs.

A Service may include the specification of the value offered to another, and the offer to provide value to another. A Service 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 Service may also be viewed as the offer of some service or set of related services provided by a provider Participant that, when consumed by some consumer Participants, has some value or achieves some objective of the connected parties. A service 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 detained 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 extended by Service.

A Service 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 Service 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 that designate the capabilities of the Participant through this Service, and the required interfaces that represent what the Participant is required 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 Service is a ServiceInterface, then the Service's provided Interfaces are the Interfaces realized by the ServiceInterface while its required Interfaces are those used by the ServiceInterface. If the type of a Service 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.

Attributes

No new attributes

Associations

No new associations

Constraints

[1] The type of a Service must be a ServiceInterface or an Interface.

[2] The direction property of a Service must be incoming.

Semantics

A Service represents a feature of a Participant through which a providing Participant with capabilities to provide a service interacts with one or more consuming participants 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 Service 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 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 Service 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 Service 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 is defined as a role within a ServiceContract, then the Service (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 «Service» keyword and/or the Service icon decoration:

Examples

Figure 6.53 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 that defines the interface to the service. See "ServiceInterface" for further details on this ServiceInterface.

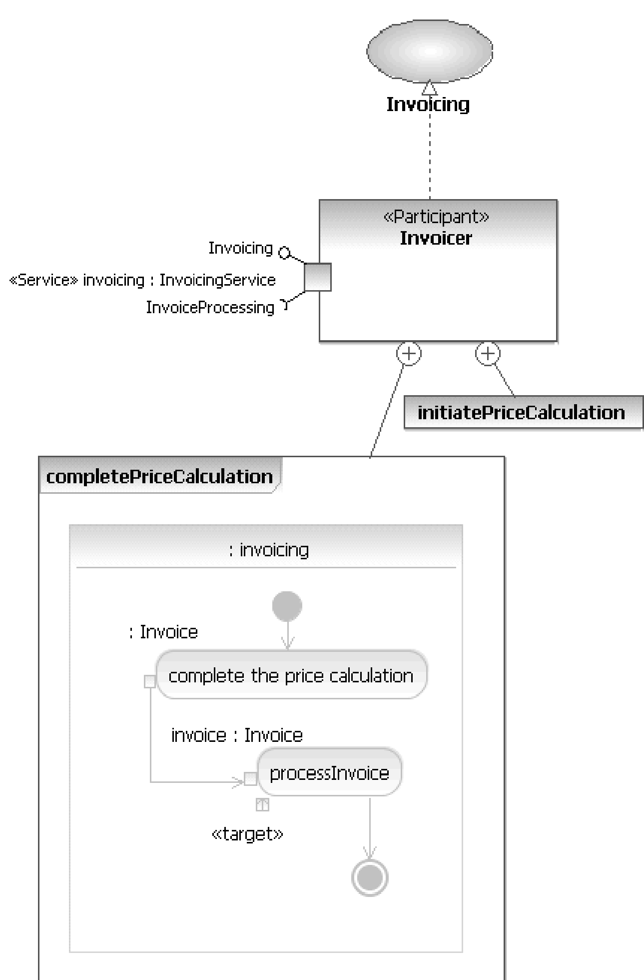


Figure 6.53 - The invoicing Service of the Invoicer Participant

The Invoicer Participant has two ownedBehaviors, one an Activity and the other an OpaqueBehavior that 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 6.54 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.

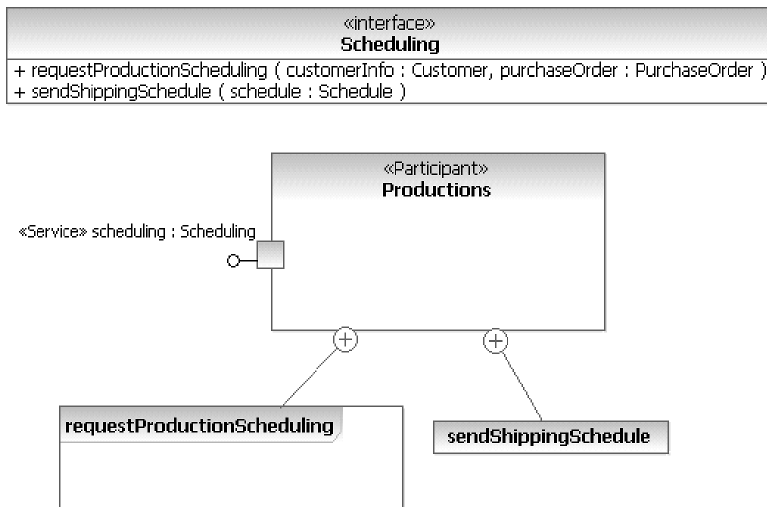


Figure 6.54 - The scheduling Service of the Productions Participant

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

Additions to UML2

None - Service is a stereotype of UML Port to designate a feature of a Participant.

6.4.19 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.

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

[1] 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.

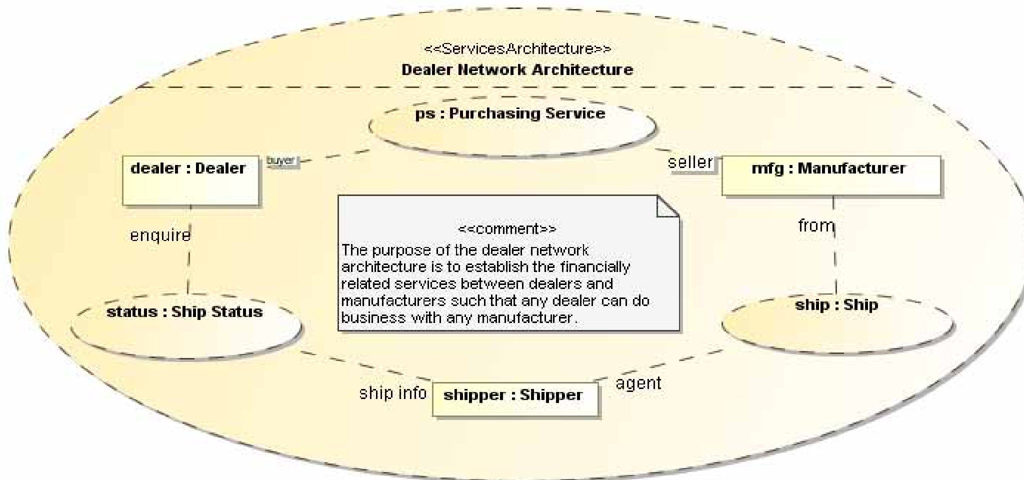


Figure 6.55 - Services architecture involving three participants

The example in Figure 6.55 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.

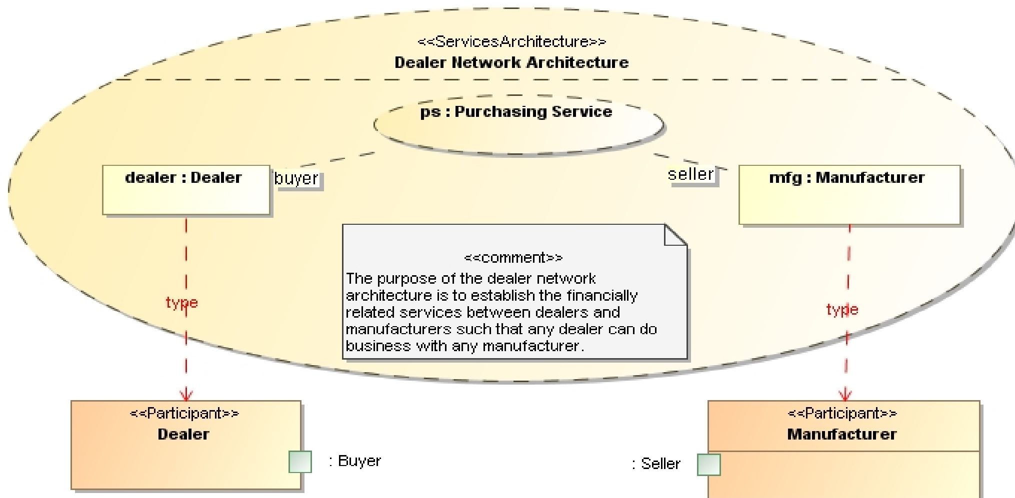


Figure 6.56 - Abbreviated service contract

The diagram in Figure 6.56 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.

Notation

A ServicesArchitecture is designated using the Collaboration notation stereotyped with «ServicesArchitecture» or using the ServicesArchitecture icon decoration:

7 Categorization

7.1 Overview

The same model may be used for many different purposes and viewed from the perspectives of many different stakeholders. As a result, the information in a model may need to be organized various ways across many orthogonal dimensions. For example, model elements might need to be organized by business domain, function, element type, owner, developer, defect rate, location, cost gradient, time of production, status, portfolio, architectural layer, Web servers, tiers in an n-tiered application, physical boundary, service partitions, etc. Another important aspect of organization is complexity management. The number and type of these different organizational schemes vary dynamically and are therefore difficult to standardize.

In addition, organizational schemes may be defined ahead of time and applied to elements as they are developed in order to characterize or constrain those elements in the organizational hierarchy. For example, biological classification schemes are used to group species according to shared physical characteristics. This technique improves consistency and enables the identification of species traits at a glance. For another example, approaches to software architecture often classify elements by their position in the architecture from user interface, controller, service, service implementation, and data tier.

Categorization not only may be used to organize and describe elements across multiple dimensions, it may also be useful for describing applicable constraints, policies, or qualities of service that are applicable to the categorized element. For example, a model element in the service layer of a Service Oriented Architecture might have additional policies for distribution, security, and transaction scope.

SoaML introduces a generic mechanism for categorizing elements in UML with categories and category values that describe some information about the elements. This mechanism is based on a subset of RAS and some stereotypes described in this section are placeholders for the similarly named elements in RAS. Categories may be organized into a hierarchy of named Catalogs. The same element may be classified by many Categories, and the same Category or CategoryValue may be applied to many elements. This is intended to be a very flexible, dynamic mechanism for organizing elements in multiple, orthogonal hierarchies for any purpose the modeler needs.

7.2 Abstract Syntax

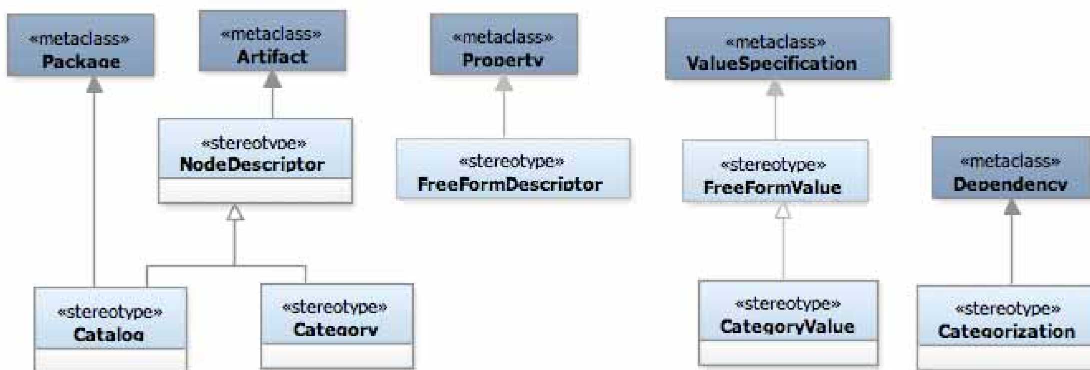


Figure 7.1 - Classification

7.3 Class Descriptions

7.3.1 Catalog

Provides a means of classifying and organizing elements by categories for any purpose.

Extends

- Package

Specializes

- NoteDescriptor

Description

A named collection of related elements, including other catalogs characterized by a specific set of categories. Applying a Category to an Element using a Categorization places that Element in the Catalog. Catalog is a RAS DescriptorGroup containing other Catalogs and/or Categories providing the mapping to RAS classification.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] Catalogs can only contain Categories, CategoryValues, or other Catalogs.

Semantics

When a model Element is categorized with a Category or CategoryValue, it is effectively placed in the Catalog that contains that Category. In the case of classification by a CategoryValue, the Category is the classifier of the CategoryValue.

The meaning of being categorized by a Category, and therefore placed in a Catalog is not specified by this specification. It can mean whatever the modeler wishes. That meaning might be suggested by the catalog and category name, the category's attributes, and a category value's attribute values. The same model element can be categorized many ways. The same category or category value may be used to categorize many model elements.

Notation

The notation is a Package stereotyped as "Catalog." Tool vendors are encouraged to provide views and queries that show elements organized in catalog hierarchies based on how they are categorized.

Figure 7.2 - Catalog Notation

7.3.2 Categorization

Used to categorize an Element by a Category or CategoryValue.

Extends

- Dependency

Description

Categorization connects an Element to a Category or CategoryValue in order to categorize or classify that element. The Element then becomes a member of the Catalog that contains that Category. This allows Elements to be organized in many hierarchical Catalogs where each Catalog is described by a set of Categories.

The source is any Element, the target is a Category or CategoryValue.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] The target of a Categorization must be either a Category or CategoryValue.

Semantics

The primary purpose of Category is to be able to provide information that characterizes an element by some domain of interest. Categorizing an element characterizes that element with that Category. What this means is derived from the meaning of the Category. The meaning of a Category is defined by its name, owned attributes, or constraints if any.

Categorization of an element may be used to provide multiple orthogonal ways of organizing elements. UML currently provides a single mechanism for organizing model elements as PackagedElements in a Package. This is useful for namespace management and any other situations where it is necessary for an element to be in one and only one container at a time. But it is insufficient for organization across many different dimensions since a PackageableElement can only be contained in one Package. For example, model elements might also need to be organized by owner, location, cost gradient, time of production, status, portfolio, architectural layer, Web, tiers in an n-tiered application, physical boundary, service partitions, etc. Different classification hierarchies and Categories may be used to capture these concerns and be applied to elements to indicate orthogonal organizational strategies.

Notation

A Category or CategoryValue may be applied to an Element Categorization that may be represented as a Dependency with the “Categorization” stereotype.

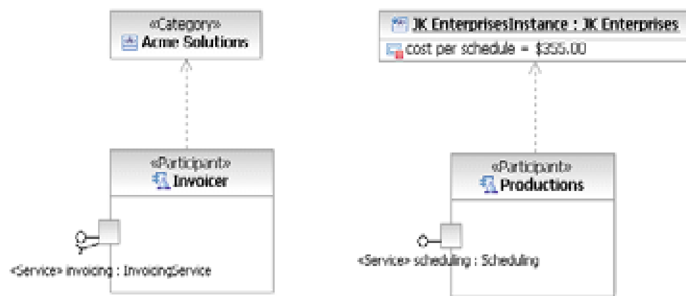


Figure 7.3 - Notation to apply a Category or CategoryValue to a model element

Changes to UML 2.1

No changes to UML 2.1

7.3.3 Category

A classification or division used to characterize the elements of a catalog and to categorize model elements.

Generalizations

- NodeDescriptor

Description

A Category is a piece of information about an element. A Category has a name indicating what the information is about, and a set of attributes and constraints that characterize the Category. An Element may have many Categories, and the same Category can be applied to many Elements. Categories may be organized into Catalogs hierarchies.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] A Category must be contained in a Catalog.

Semantics

The meaning of a Category is not specified by SoaML. Instead it may be interpreted by the modeler, viewer of the model, or any other user for any purpose they wish. For example a Catalog hiearachy of Categories could be used to indicate shared characteristics used to group species. In this case the categorization might imply inheritance and the principle of common descent. Other categorizations could represent some other taxonomy such as ownership. In this case, the term categorization is intended to mean describing the characteristics of something, not necessarily an inheritance hierarchy. All instances having categorized by a Category have the characteristics of that Category.

The characteristics of a Category are described by its attributes and constraints. ClassifierValues may be used to provide specific values for these attributes in order to more specifically categorize an element.

A Category may have ownedRules representing Constraints that further characterize the category. The meaning of these constraints when an element is categorized by a Category is not specified.

Notation

The notation is an Artifact stereotyped as "Category."

Figure 7.4 - Category Notation

Examples

Ownership

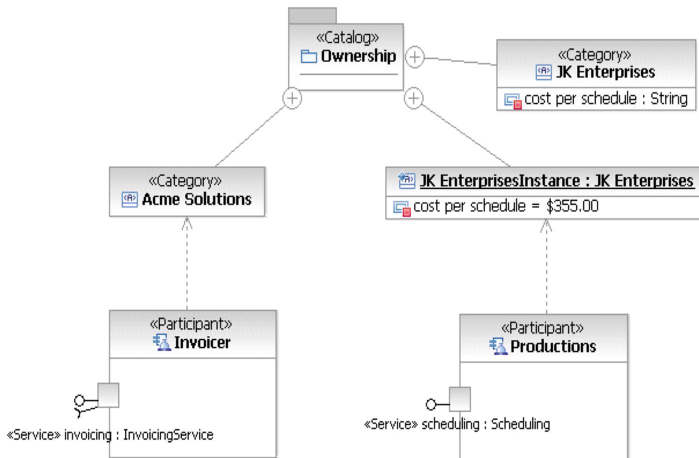


Figure 7.5 - Ownership

Figure 7.5 shows a simple example of using Categories to denote ownership of service participants. Categories Acme Solutions and JK Enterprises are contained in a Catalog called Ownership. Participant Invoicer is owned by Acme Solutions. Participant Productions is owned by JK Enterprises. The cost per schedule for Productions is $355.00. Note that the SoaML model does not know the meaning of Acme Solutions, or that the Category denotes ownership. Nor does the model know what cost per schedule means or how it is applied. Tools that manipulate the model provide facilities for accessing the categories that categorize a model element and its values in order to produce reports, effect transformations, provide model documentation, or any other purpose.

7.3.4 CategoryValue

Provides specific values for a Category to further categorize model elements.

Generalizations

- FreeFormValue

Description

A CategoryValue provides values for the attributes of a Category. It may also be used to categorize model elements providing detailed information for the category.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] The classifier for a CategoryValue must be a Category.

Semantics

The characteristics of a Category are described by its attributes and constraints. ClassifierValues may be used to provide specific values for these attributes in order to more specifically categorize an element.

Categorizing an element with a CategoryValue categorizes the element by the Category that is the classifier of the CategoryValue.

Notation

The notation is an InstanceSpecification stereotyped as «CategoryValue».

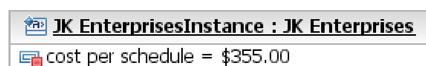


Figure 7.6 - CategoryValue Notation

7.3.5 RAS Placeholders

The following stereotypes represent placeholders for the corresponding elements in the OMG Reusable Asset Specification (RAS). These placeholders are included to provide SoaML integration with RAS. For further details, see the RAS specification (http://www.omg.org/spec/RAS/).

- NodeDescriptor extends Artifact
- FreeFormDescriptor extends Property
- FreeFormValue extends ValueSpecification

There are some differences between SoaML categorization and RAS:

- RAS FreeFormValues are contained in a ClassificationSchema and may be used individually to classify any asset. SoaML uses Category ownedAttributes to define Properties of a Category. These Properties are encapsulated in a Category and cannot be used in another Category.
- RAS uses a DescriptorGroup to associate a ClassificationSchema and set of FreeFormValues of FreeFormDescriptors from that ClassificationSchema to classify an Asset. SoaML uses Categorization Dependencies to categorize any model Elements.

8 BMM Integration

8.1 Overview

The promise of SOA is to provide an architecture for creating business relevant services that can be easily reused to create new business integration solutions. In order to indicate a service's business relevance, they may be linked to business motivation and strategy. SoaML provides a capability for connecting to OMG Business Motivation Model (BMM) models to capture how services solutions realize business motivation. This connection to BMM is optional and assumes BMM is provided either as a UML profile, or as a package that can be merged with SoaML.

8.2 Abstract Syntax

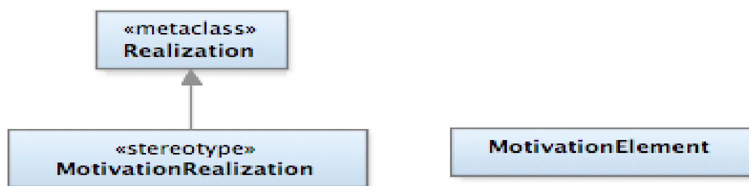


Figure 8.1 - The BMM Integration Profile Elements

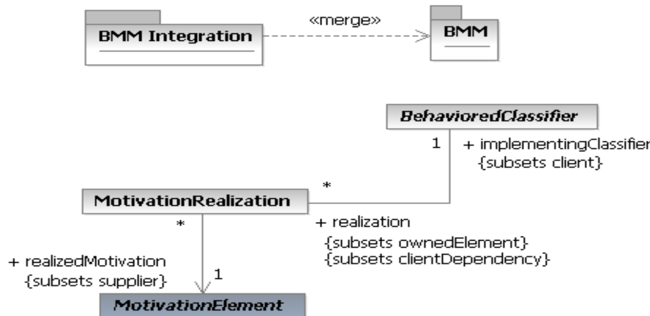


Figure 8.2 - BMM Integration Package

8.3 Class and Stereotype Descriptions

8.3.1 MotivationElement

Generalizations

Extensions

Description

A placeholder for BMM MotivationElement. This placeholder would be replaced by a BMM profile or metamodel element.

8.3.2 MotivationRealization

Generalizations

- Realization

Extensions

- Realization

Description

Models a realization of a BMM MotivationElement (a Vision, Goal, Objective, Mission, Strategy, Tactic, BusinessPolicy, Regulation, etc.) by some BehavioredClassifier.

Attributes

- No additional attributes

Associations

- realizedEnd: End [*]
  The ends realized by this MeansRealization. (Metamodel only)

Constraints

No additional constraints

Semantics

Notation

MotivationRealization uses the same notation as Realization in UML2. The source and targets of the Realization indicate the kind of Realization being denoted.

Additions to UML2

No

Examples

Figure 8.3 shows an example of a business motivation model that captures the following business requirements concerning the processing of purchase orders:

- Establish a common means of processing purchase orders.
- Ensure orders are processed in a timely manner, and deliver the required goods.
- Help minimize stock on hand.
- Minimize production and shipping costs

This example of a BMM model shows the business vision, the goals that amplify that vision, and the objectives that quantify the goals. It also shows the business mission, the strategies that are part of the mission plan, and the tactics that implement the strategies. Finally the strategies are tied to the goals they support.

The example also shows a Process Purchase Order contract that formalizes the requirements into specific roles, responsibilities, and interactions. The Contract indicates what motivation elements it realizes through MeansRealizations.

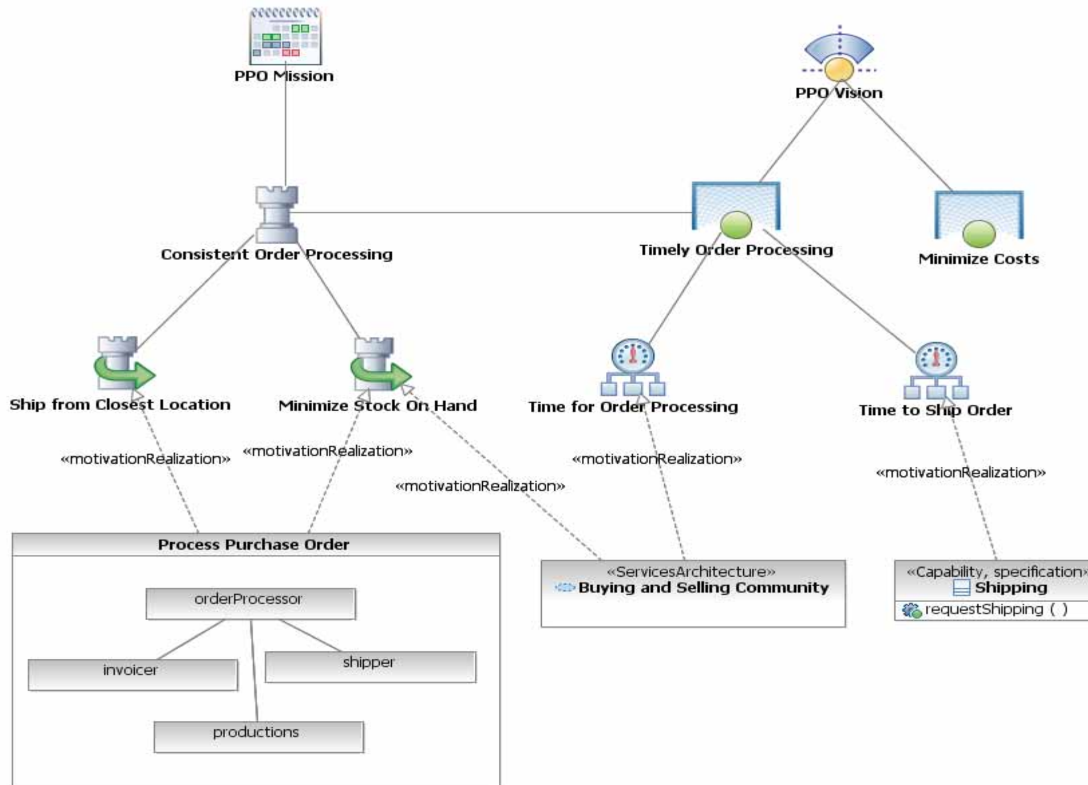


Figure 8.3 - Business Motivation Model for Processing Purchase Orders

Additions to UML2

Adds a link between UML and BMM that exploits Collaboration and CollaborationUse to provide a semantically rich way of indicating how requirements captured in a business motivation model are realized and fulfilled by elements in a UML model.

9 SoaML Metamodel

9.1 Overview

The SoaML Metamodel extends the UML2 metamodel to support an explicit service modeling in distributed environments. This extension aims to support different service modeling scenarios such as single service description, service oriented architecture modeling, or service contract definition.

The metamodel extends UML2 in five main areas: Participants, Services, Interfaces, Service Contracts, and Service data. The participants enable us to define the service providers and consumers in a system. ServiceInterfaces make it possible to explicitly model the operation provided and required to complete the functionality of a service. The ServiceContracts are used to describe interaction patterns between service entities. Finally, the metamodel also provides elements to model service messages explicitly and also to model message attachments.

Figure 9.1 illustrates the elements that support the Participants and the ServiceInterface modeling. A Participant may play the role of service provider, consumer, or both. When a participant works as a provider it contains Service Points. On the other hand, when a participant works as a consumer it contains Request Ports.

A ServiceInterface can be used as the protocol for a Service or a Request Port.

- If it is used in a Service Port, it means that the Participant who owns the ports is able to implement that ServiceInterface.
- If it is used in a Request Port, it means that the Participant uses that ServiceInterface.

The "Makes" relation is derived from "ownedPort" where the port is a Request.

The "offers" relation is derived from "ownedPort" where the port is a Service.

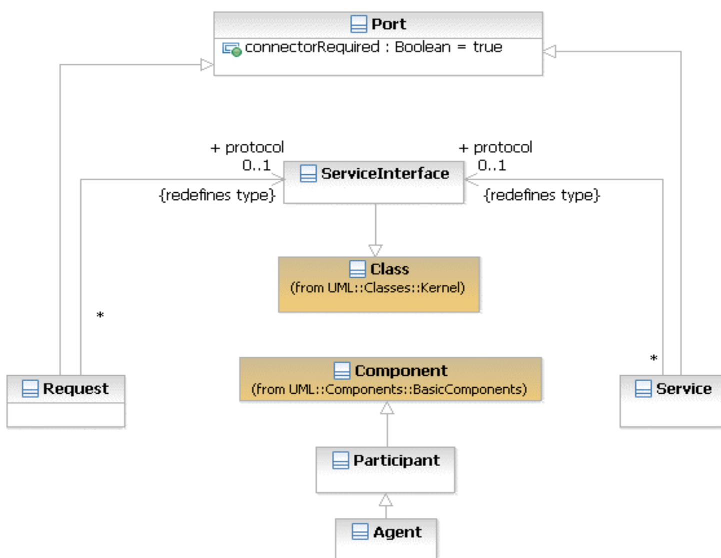


Figure 9.1 - ServiceInterfaces and Participants

Figure 9.2 presents the elements that support the ServiceContract Modeling. These contracts can later be realized by service elements such as Participants, ServiceInterfaces or ConnectableElements (Request or Service Ports). Another element included in this figure is the ServiceArchitecture. The ServiceArchitecture is used to model service architectures and their owned behaviors.

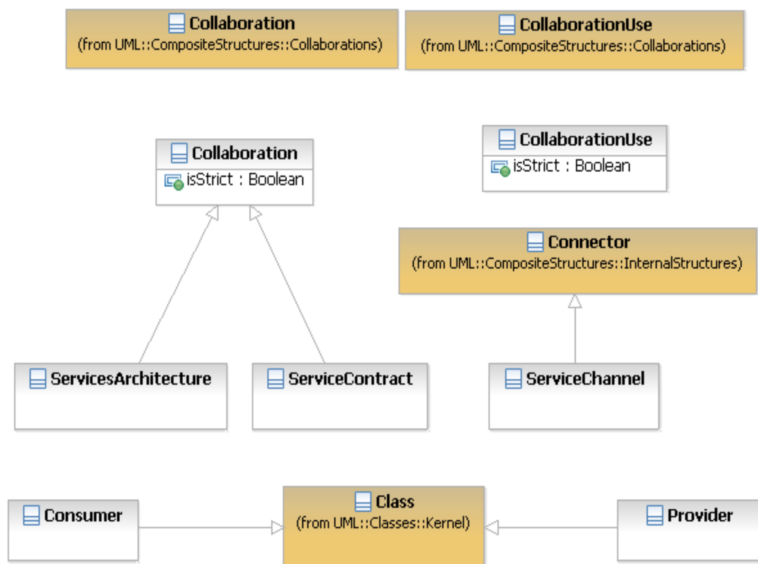


Figure 9.2 - ServiceContracts and ServiceArchitectures

Figure 9.3 presents the elements that support the Data Modeling. Attachments are used to model elements that have their own identity when they are taken out of the system. For example we can define an attachment to send a text document that can be used with other applications external to the system. The MessageType is used to explicitly identify data elements that will travel among the participants of the service interaction.

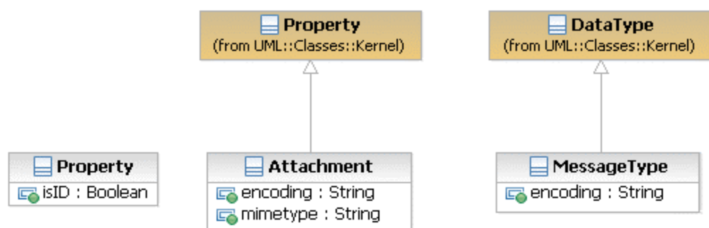


Figure 9.3 -Service Data

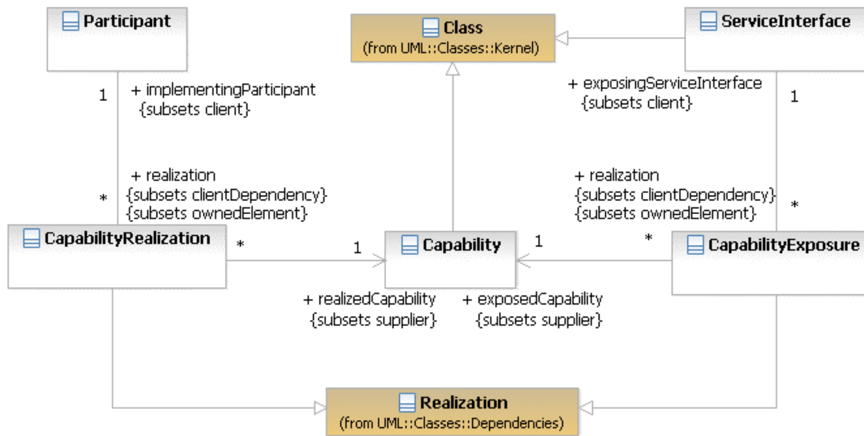


Figure 9.4 - Capabilities

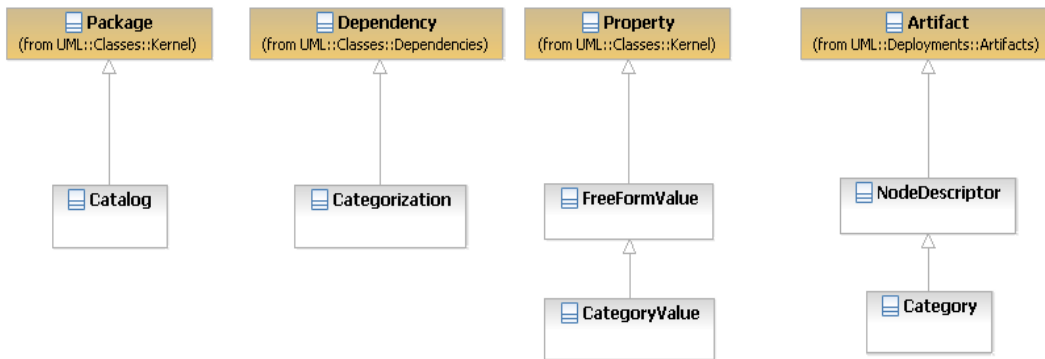


Figure 9.5 - Categorization

9.2 Profile metamodel mapping

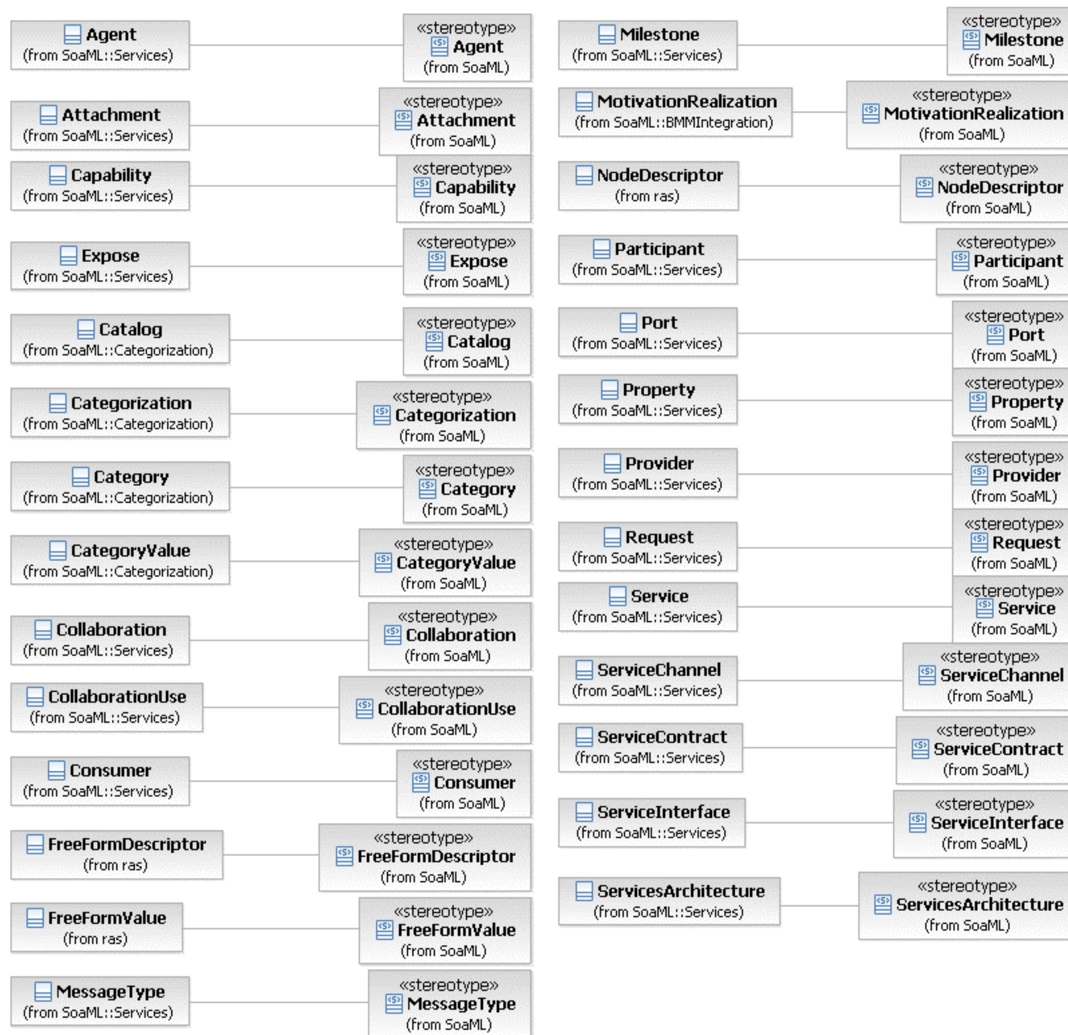


Figure 9.6 - Profile metamodel mapping

Annex A: Relationship to OASIS Services Reference Model

(informative)

This specification attempts to leverage existing work by OASIS and others to ensure compatibility with existing reference models and Web Services platforms. The initial OASIS Reference model for Service Oriented Architecture (version 1.0, Oct. 2006) has been followed up by a broader OASIS Reference Architecture for SOA in April 2008. This work was provided by Jeff Estefan, an editor of the OASIS RA document.

Recently also the Open Group has published a draft SOA Ontology. (July 2008).

In the following table we compare the definition of main concepts of SoaML with the definition of the similar concepts in the other reference models.

	SoaML	**SOA-RM**	**SOA-RA**	**SOA Ontology**
Org	OMG	OASIS	OASIS	The Open Group
Version	1.8 - Revised Submission	1.0	1.0 – Pubic Review Draft 1	*Not identified*
Date	Aug 25, 2008	Oct 12, 2006	Apr 23, 2008	Jul 14, 2008
Status	Draft Standard	Completed Standard	Draft Specification	Draft Standard
Concept	**Definition**	**Definition**	**Definition**	**Definition**
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.	*Not explicitly defined.*	Any entity that is capable of acting on behalf of a person or organization.	*Not explicitly defined*
Collaboration	Collaboration from UML is extended to describe ServiceContracts and ServicesArchitecturesServices Architectures.	*Interaction:* The activity involved in making using of a capability offered, usually across an ownership boundary, in order to achieve a particular desired real-world effect.	*Adopts SOA-RM definition*	
CollaborationUse	CollaborationUse shows how a Collaboration (ServiceContracts and ServiceArchitectures) is fulfilled.			

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.	*Not explicitly defined*	*Not explicitly defined*	*Not explicitly defined*
Participant	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 or component.	*Not explicitly defined.*	A stakeholder that has the capability to act in the context of a SOA-based system. See also Service Provider and Service Consumer below.	
Real World Effect	Defined as "service operation post condition."	The actual result of using a service, rather than merely the capability offered by a service provider.	*Adopts SOA-RM definition*	Defined as Effect. Comprises the outcome of performance of the service, and is the value delivered.
Request Port (port stereotype)	A request port defines the port through which a Participant makes requests and uses or consumes services.			
Service Port (port stereotype)	The service port stereotype of a port defines the connection point the point of interaction on a Participant where a service is actually provided or consumed.			

Service (general)	*Service* is defined as a resource that enables access to one or more capabilities. Here, the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. This access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. A service is provided by an entity—called the *provider*—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.	A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.	*Adopts SOA-RM definition*	A logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit; provide weather data, consolidate drilling reports). It is self-contained, may be composed of other services, and is a "black box" to its consumers.
Capability	Identifies or specifies a cohesive set of functions or capabilities that a service provides.			
Capability	The ability to act and produce an outcome that achieves a result. As such, capability involves the capacity, power, or fitness for some specified action or operation. This implies that the entity must have physical, mental, or legal power to generate an outcome that achieves a real world effect. (synonymous with capability) specifies a A Capability models the capability for providing, or provided by, a service specified by a ServiceContract or ServiceInterface.			

Service Contract	A ServiceContract is the formalization of a binding exchange of information, goods, or obligations between parties defining a service. 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 or implementation.	A **contract**, represents an agreement by two or more parties. A service contract is a measurable assertion that governs the requirements and expectations of two or more parties.	*Adopts SOA-RM definition*	*Adopts SOA-RM definition*
Service Interface	Defines the interface to a Service or Request. 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 port. A ServiceInterface is the means for specifying how to interact with a Service.	*Service Description* The information needed in order to use, or consider using, a service.	*Adopts SOA-RM definition.*	*Description*. An information item that is represented in words, possibly accompanied by supporting material such as graphics. The **Description** class corresponds to the concept of a description as a particular kind of information item that applies to something in particular – the thing that it describes. It is not just a set of words that could apply to many things.

ServiceChannel	A communication path between Requests and services. A ServiceChannel provides a communication path between consumer Requests (ports) and provider services (ports).			
Service Oriented Architecture, Services Architecture	An architectural paradigm for defining how people, organizations and systems provide and use services to achieve results. *Services Architecture* The high-level view of a Service Oriented Architecture that defines how a set of participants works together for some purpose by providing and using services. A Services Architecture (an SOA) describes how participants work together for a purpose by providing and using services expressed as service.	A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.	*Adopts SOA-RM definition*	An architectural style that supports service orientation. An *architectural style* is the combination of distinctive features in which architecture is performed or expressed.

The Conformance Guidelines of the OASIS Reference Model for SOA, section 4, outlines expectations that a design for a service system using the SOA approach should meet:

- Have entities that can be identified as services as defined by this Reference Model;
    - Be able to identify how visibility is established between service providers and consumers;

      This SoaML specification defines a Service metaclass, a kind of UML Port, which establishes the interaction point between service consumers and providers. A Service's type is a ServiceInterface that provides all the information needed by a consumer to use a service. However, the UPMS RFP specifies that mechanisms for discovering existing services and descriptions consumers would use to determine the applicability of availability of existing services for their needs (awareness) are out of scope and are therefore not covered in this specification.

    - Be able to identify how interaction is mediated;

      Interaction between a service consumer and provider connected through a service channel is mediated by the protocol specified by the service provider. The protocol is defined by the service interface used as the type of the service and may include a behavior that specifies the dynamic aspects of service interaction. The interfaces

realized and used by a service specification define the operations, parameters, preconditions, post conditions (real world effect), exceptions and other policy constraints that make up the static portion of the service specification.

- Be able to identify how the effect of using services is understood;

  The effect of a service is specified by the post conditions of the provided service operations assuming the consumer follows the policies, preconditions, and protocols specified by the service interface.

- Have descriptions associated with services;

  This specification includes a service interface for describing the means of interacting with a service. Since service discovery and applicability are out of scope for the UPMS RFP, this specification does not include additional description information a consumer might need to consider using a service.

- Be able to identify the execution context required to support interaction;

  The execution context is specified by the semantics for UML2 as extended by this specification.

It will be possible to identify how policies are handled and how contracts may be modeled and enforced.

Policies are constraints that can be owned rules of any model element, including in particular service ports and service participant components. The actual form of these policies is out of scope for the UPMS RFP and are not further developed in this specification.

We have also collected other definitions around services and SOA and are analyzing this with respect to further need for harmonization between the standardization groups, in particular for the use of the concept service.

Annex B: Examples

(informative)

This Annex provides examples of a SOA model to establish a foundation for understanding the submission details. Service modeling involves many aspects of the solution development lifecycle. It is difficult to understand these different aspects when they are taken out of context and explained in detail. This example will provide the overall context to be used throughout the specification. It grounds the concepts in reality, and shows the relationships between the parts. The example is elaborated in other sections in order to explain specification details. The stereotypes used describe the minimum notation extensions necessary to support services modeling. These stereotypes may be viewed as either keywords designating the notation for the UPMS metamodel elements, or stereotypes defined in the equivalent UPMS profile.

The examples are broken into three parts:

1. The dealer network architecture that defines a B2B community, facilitated with SOA.
2. A purchase order process example for a member of this community that develops internal services in support of that process.
3. A sample purchase order data model appropriate to both the community and process examples.

B.1 Dealer Network Architecture

B.1.1 Introduction

Our example business scenario is a community of independent dealers, manufacturers, and shippers who want to be able to work together cohesively and not re-design business processes or systems when working with other parties in the community. On the other hand they want to be able to have their own business processes, rules, and information. The community has decided to define a service oriented architecture for the community to enable this open and agile business environment.

B.1.2 Defining the community

The dealer network is defined as a community “collaboration” involving three primary roles for participants in this community: the dealer, manufacturer, and shipper. Likewise the participants participle in three “B2B” services - a purchasing service, a supping service, and a ship status service. The following diagram illustrates these roles and services in the dealer network architecture.

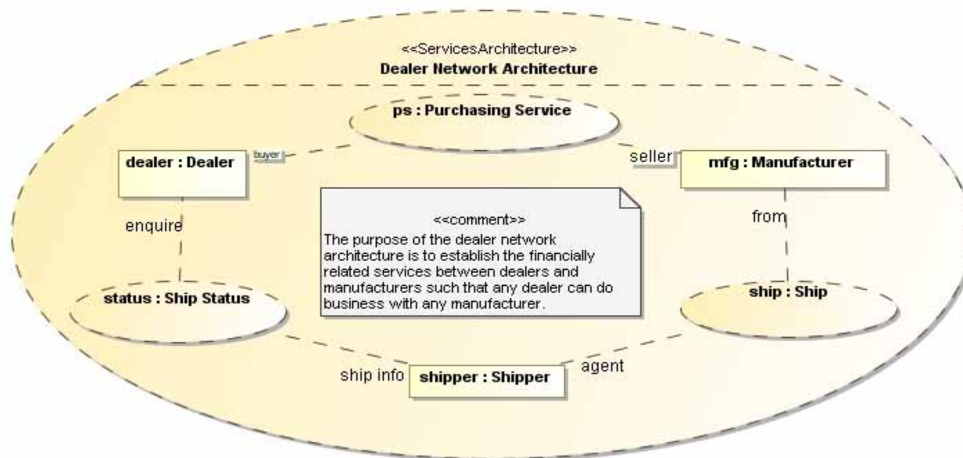


Figure B.1 - Dealer network architecture

Note that there is a direct correspondence between the roles and services in the business community and the SOA defined as a SoaML community collaboration "ServicesArchitecture." The services architecture provides a high-level and contextual view of the roles of the participants and services without showing excessive detail. Yet, the detail is there-as we shall see as we drill down. Note that the detail is fully integrated with the high-level view providing traceability from architecture through implementation.

One additional detail that can be seen in this diagram is the roles that the participants play with respect to each of the services. Note that the manufacturer "plays the role" of "seller" with respect to the purchasing service and the dealer "plays the role" of the "buyer" with respect to the same service. Note also that the manufacturer plays the "from" role with respect to the "Ship" service. It is common for participants to play multiple roles in multiple services within an architecture. The same participant may be "provider" of some services and a "consumer" of other.

There are various approaches to creating the services architecture - some more "top down" and others more "bottom up." Regardless of the "direction" the same information is created and integrated as the entire picture of the architecture evolves. For this reason we will avoid implying that one part is "first," the goals is to complete the picture - filling in details as they are known, based on the methodology utilized.

Since a B2B architecture is among independent participants there is usually no "business process" other than the services. However a business process may be defined for a community if one is required, or for any of the participants (for example, for a community within an organization may have a specific process). When defining the services architecture for a participant a business process is frequently specified.

B.1.3 Services to support the community

Considering the "Purchasing Service" the service in the Services Architecture does not define the service, it uses the service specification. The service is defined independently and then dropped into the service architecture so it can be used and reused. The purchasing service is actually a composite service like many enterprise services, as can be seen in Figure B.2.

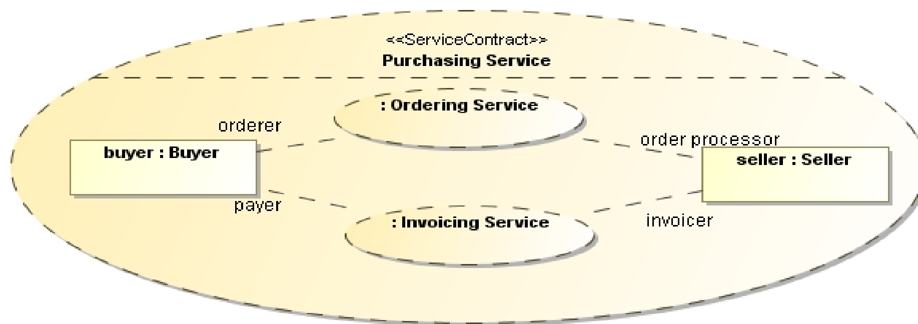


Figure B.2 - Purchasing service

The diagram above shows that the purchasing service is actually composed of two simpler services: Ordering service and Invoicing Service. Of course a real enterprise service would probably have many more sub-services. In this scenario, the "buyer" is the "orderer" of the order processing service and the "payer" of the invoicing service. The seller is the "order processor" of the ordering service and "invoicer" of the invoicing service.

Looking at the "Ordering" service in more detail we will identify the roles of participants in that service as shown in Figure B.3.

Figure B.3 - Ordering service

This diagram simply identifies the "Service Contract," the terms and conditions of "ordering," as well as defining the two roles: Orderer and Order Processor. We then want to add some more detail describing the flow of information (as well as products, services, and obligations) between the participants. This is done using a UML behavior as shown in Figure B.4.

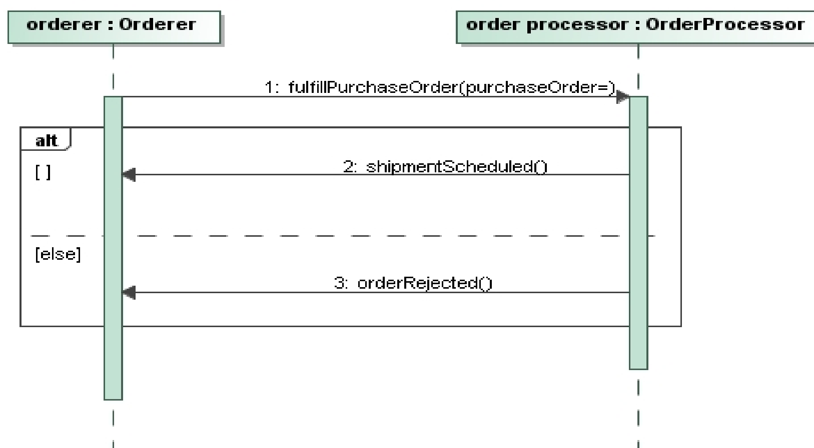


Figure B.4 - Flow of information

This is the “behavior” associated with the OrderingService service contract. This behavior is then required of any participant who plays a role in these services. The service contract is “binding” on all of the parties. The behavior shows how the participants work together within the context of this service - not their internal processes. The specific behavior here is how the messages are “choreographed” in the service contract - what flows between who, when, and why.

Note that the service contract behavior shows what information flows between the participants (such as PurchaseOrder and ShipmentSchedule) and also defines when these interactions take place. This is the “choreography” of the service contract; the choreography defines what flows between the parties, when, and under what conditions. Rules about the services are frequently attached to the choreography as UML constraints.

This behavior is quite simple: the orderer sends a “fulfillPurchaseOrder” to the OrderProcessor and the orderProcessor sends back either a “shipmentSchedule” or an “orderRejected.” In this diagram we don’t see the details of the message content, but that details is within the model as the arguments to these messages.

B.1.3.1 Inside of a manufacturer

It is architecturally significant and desirable that the services architecture not detail the business processes or internal structure of each of the participants - this allows each party maximum freedom in how they achieve their goals without overly constraining how they do it. If everyone’s processes were pre-defined, the entire architecture would become fragile and overly constraining. Limiting the community’s specification to the contract of services between the parties provides for the agile environment that is the hallmark of SOA.

While it is important to not over-specify any of the parties at the level of the community, it is equally important for a particular manufacturer to be able to create or adapt their internal architecture in a way that fulfills their responsibilities within the community. So we want to be able to “drill down” into a particular manufacturer and see how they are adapting to these services.

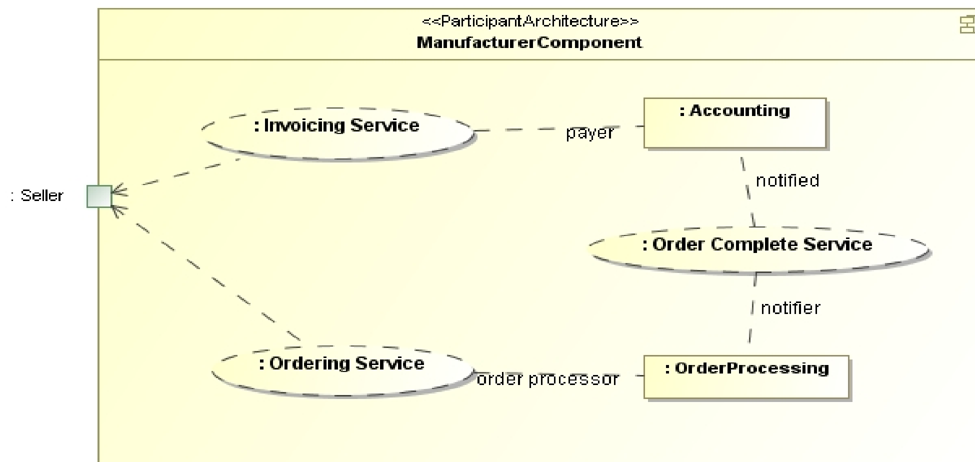


Figure B.5 - Manufacturer component

The diagram above shows the participant services architecture of a manufacture that complies with the community architecture. Note that the architecture of a participant is defined for the manufacture as a "composite structure," this is one of the new features of UML-2. While it is a composite structure we can also see that the pattern of participants playing roles and services being used is the same. In this case the manufacturer has "delegated" the invoicing service and ordering service to "accounting" and "order processing," respectively. Accounting and Order Processing are participants, but participants acting within the context of the Manufacturer. Since they are operating within this context it also makes sense to define a business process for the manufacturer with the participants as swim lanes, as shown in Figure B.6.

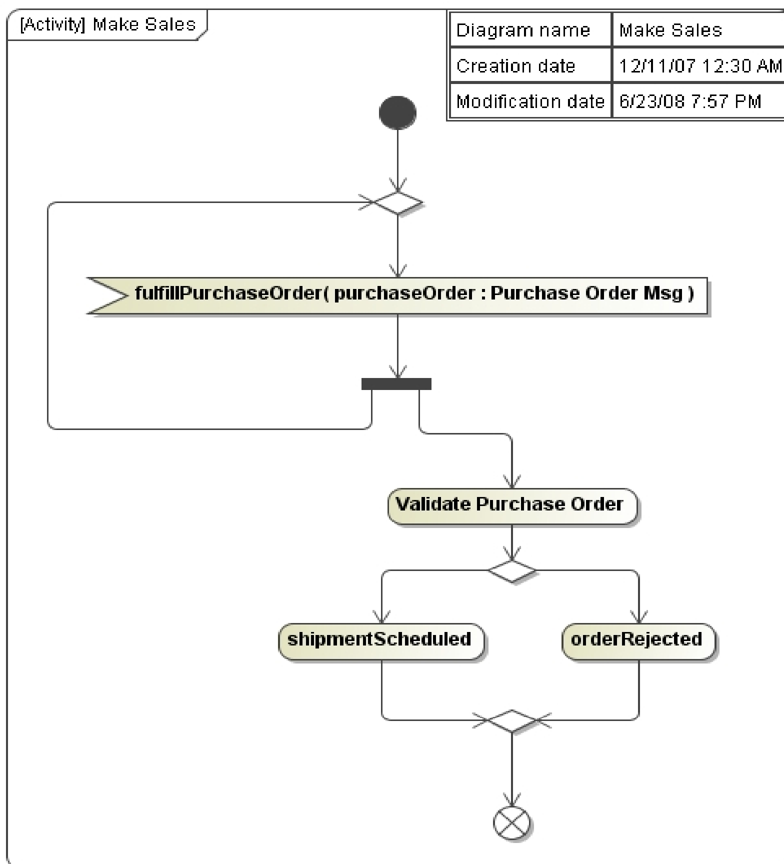


Figure B.6 - Make sales business process

Note that for brevity of the example this is just a portion of the business process but it shows how the information flows of the SOA can correspond to activities within a participant.

B.1.3.2 From services architectures and contracts to components

Everything we have looked at so far has been very high-level and could apply equally well to a pure business model or a "system of systems" model. To detail the model more, such that it can be realized in technology, we may want to define "components" that correspond to each of the roles and responsibilities we have defined. A "component" in this regard can be a business component (as a department is a component of a company) or a system component implemented in some runtime technology. In either case we would like to "drill down" to sufficient detail such that, with the appropriate technical platform and decisions, the architecture can be executed - put into action. In the case of technology components this means making the components defined in the SOA a part of the runtime application.

Participants

A participant is a class that models an entity that participates in a process according to the design laid out in the SOA. The participant has service ports that are the connection points where the services are actually provided or consumed.

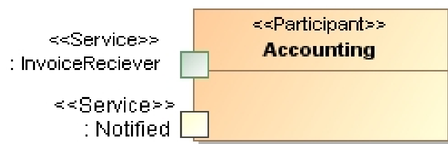


Figure B.7 - Accounting participant

The class above is an example of a participant. Since this "Accounting" participant plays a role in the manufacturer component, above, and plays a role with respect to two services - it has a service port for each one of the services it plays a role in. In this case "InvoiceReceiver" as its role in the "Invoicing" service a "Notified" in its role in the shipping service. Likewise there is a participant component "behind" each role in a services architecture. A participant may play a role in multiple services architectures and therefore must have ports that satisfy the requirements of each.

Service Interfaces

The service ports of the participants have a type that defines the participants responsibilities with respect to a service - this is the "ServiceInterface." The service interface is the type of a role in one and only one service contract. The service contract details the responsibilities of all of the participants- their responsibilities with respect to a service. The service interface is the "end" of that service that is particular to a specific participant. Each service interface has a type that realizes the interfaces required of that type as well as using the services that must be supplied by a consumer of that service (since many services are bi-directional).

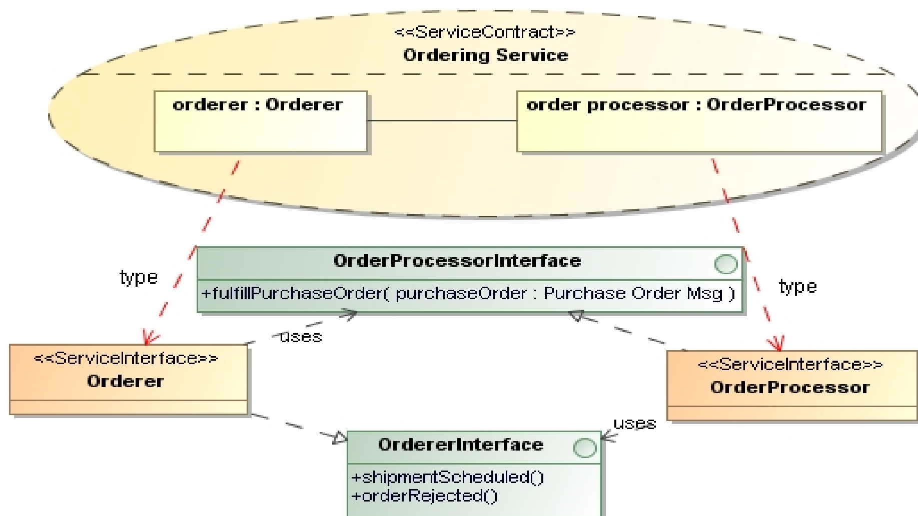


Figure B.8 - Service interfaces

The diagram, above, shown how the "Service Interfaces" are related both to the UML interfaces that define the signals and operations implemented, as well as those used. This common pattern defines a bi-directional asynchronous service. Note that the service interfaces are the types of the roles in the service contracts. These same service interfaces will be the types of the service ports on participants - thus defining the contracts that each participant is required to abide by.

Relating services architecture to service contracts

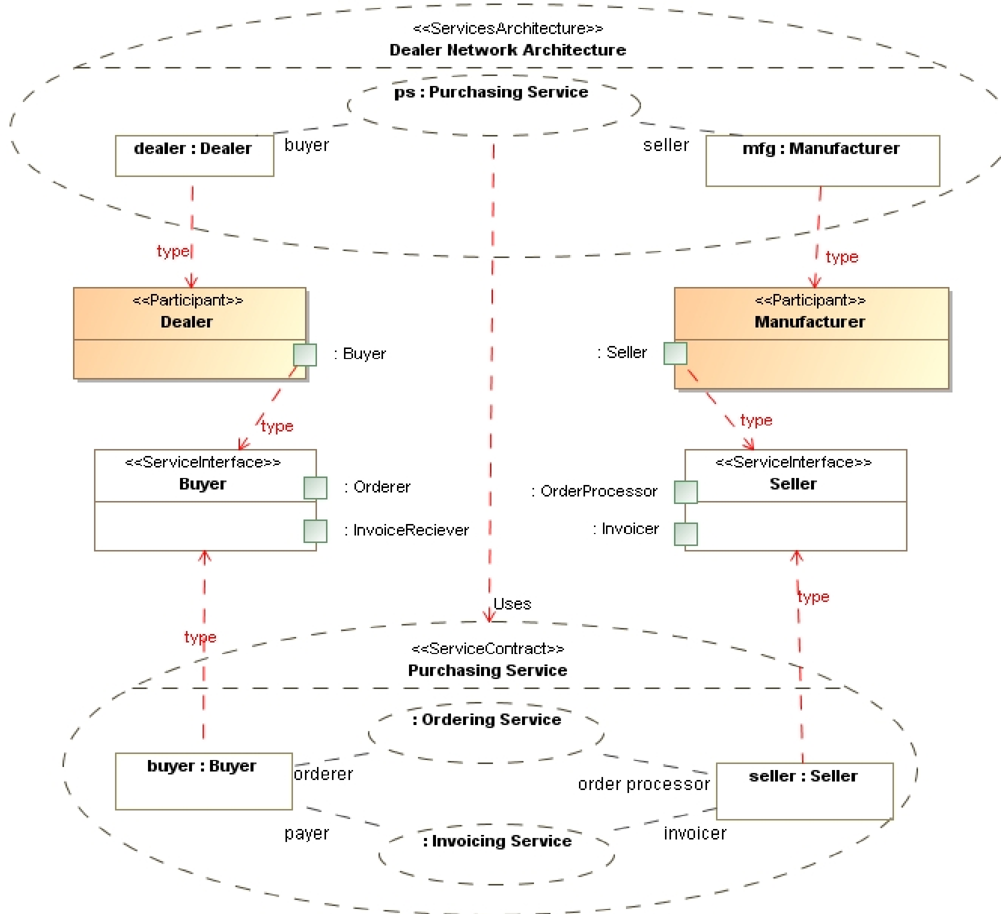


Figure B.9 - Relating services architecture as service contracts

The diagram, above, shown the “trace” between a service architecture through the participants and their service ports to the service contracts that defines the service interfaces for each of the service ports. Note that the lines in red are just to show how the underlying UML relationships are defined and are not part of the architecture.

What the above shows is that the Dealer & Manufacturer are participants in a “Dealer Network Architecture” in which the dealer plays the role of the “buyer” in the “PurchasingService” and the “Manufacturer” plays the role of the seller in the same service. Participating in these services requires that they have service ports defined on the participant type - these are the ports on “Dealer” and “Manufacturer” types. These ports have a ServiceInterface type defined in the “Purchasing Service” contract. These service interfaces each have two ports because Purchasing Service is a compound service contract - the service interfaces have a port for each nested service: OrderService & InvoicingService, respectively.

Further detail for interfaces, operations and message data

Further details on the interfaces and message structures are not reproduced in this document, since these are represented by standard and well known UML constructs.

Purchase Order Process Example

This example is based on the Purchase Order Process example taken from the UPMS RFP, which was based on an example in the BPEL 1.1 specification. It is quite simple, but complex and complete enough to show many of the modeling concepts and extensions defined in this specification. The following shows another example approach for defining services using SoaML.

The requirements for processing purchase orders are captured in a Collaboration. This example does not cover how this collaboration was determined from business requirements or business processes, rather it shows the results of requirements analysis using a simple collaboration. The collaboration is then fulfilled by a number of collaborating Participants having needs and capabilities through request and service ports specified by ServiceInterfaces identified by examining the collaboration. SoaML supports a number of different ways to capture and formalize services requirements including ServicesArchitectures, Capability usage hierarchies, or ServiceContracts. This example uses a collaboration in order to keep the example simple and to focus on the resulting ServiceInterfaces, Participants, and participant assemblies that would be typically seen in Web Services implementations using XSD, WSDL, SCA, and BPEL. Regardless of the approach used to discover and constrain the services, these participants would reflect a reasonable services analysis model.

B.1.3.3 The Services Solution Model

Representing Service Requirements

As a precursor to defining services some methodologies may use various techniques to analyze requirements leading to a full SOA. The requirements for how to processing purchase orders can be captured in a simple collaboration indicating the participating roles, the responsibilities they are expected to perform, and the rules for how they interact. This collaboration could have been derived or created from a BPMN business process and represented as a services architecture. It is treated as a formal, architecturally neutral specification of the requirements that could be fulfilled by some interacting service consumers and providers, without addressing any IT architecture or implementation concerns. It is architecturally neutral in that it does not specify the participants, or their services or requests, or any connections between the participants - only the roles they play. The services architecture will eventually result in the connections between participants with requests representing needs and participants with services that satisfy those needs.

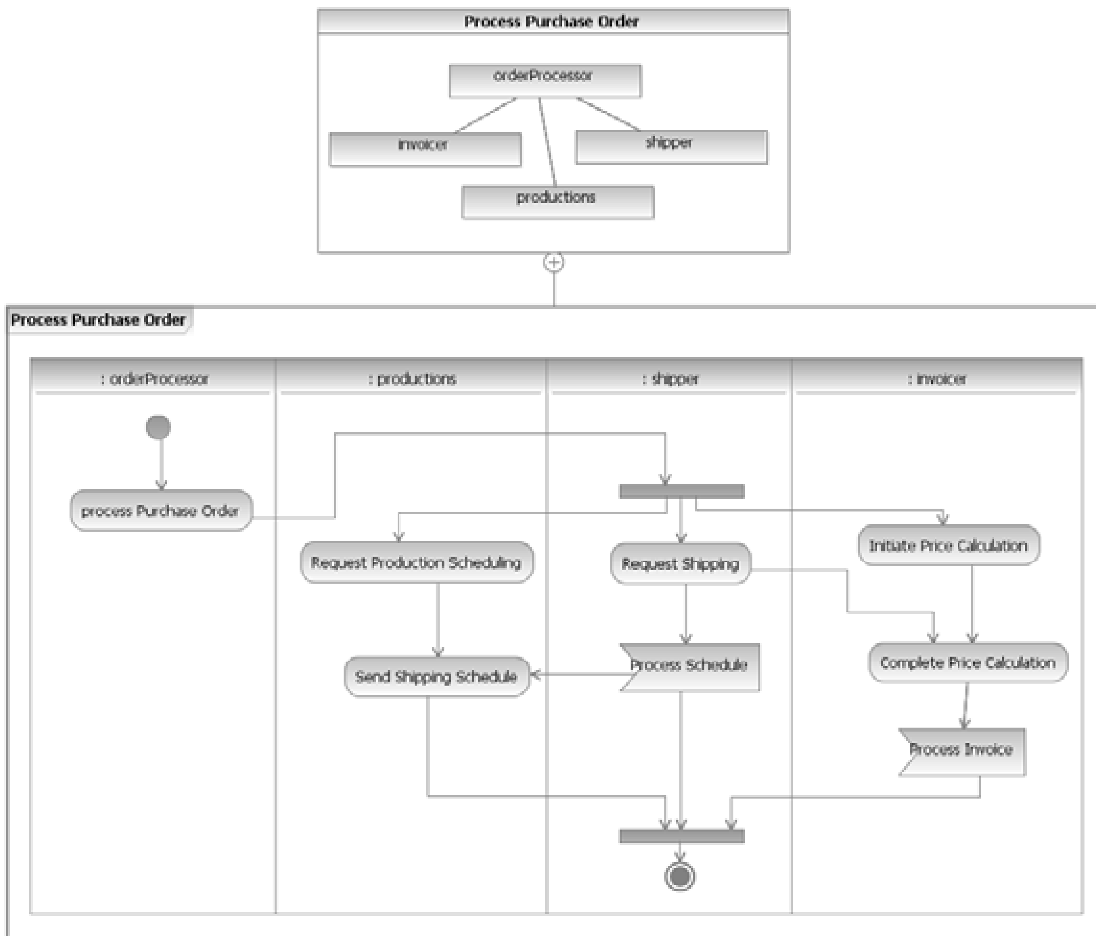


Figure B.10 - Process Purchase Order Requirements

The Process Purchase Order collaboration in Figure B.10 indicates there are four roles involved in processing purchase orders. The orderProcessor role coordinates the activities of the other roles in processing purchase orders. The types of these roles are the Interfaces shown in Figure B.11. These Interfaces have Operations that represent the responsibilities of these roles. The Process Purchase Order Activity owned by the collaboration indicates the rules for how these roles interact when performing their responsibilities.

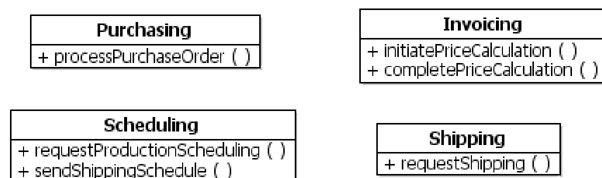


Figure B.11 - Interfaces Listing Role Responsibilities

The collaboration may have constraints indicating objectives it is intending to accomplish, how to measure success, and any rules that must be followed.

Service Identification

The next step in this example service methodology is to examine the collaboration and identify services and participants necessary to fulfill the indicated requirements. Eventually a service provider will be designed and implemented that is capable of playing each role in the collaboration, and providing the services necessary to fulfill the responsibilities of that role.

Figure B.12 shows a view of the service interfaces determined necessary to fulfill the requirements specified by the collaboration in Figure B.10. This view simply identifies the needed service interfaces, the packages in which they are defined, and the anticipated dependencies between them.

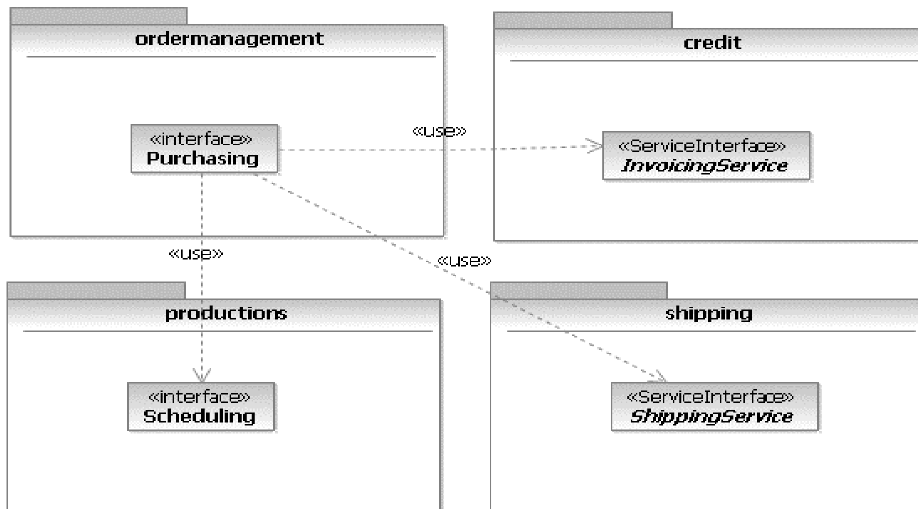


Figure B.12 - Identified Service Interfaces

Service Specification

The identified ServiceInterfaces must now be defined in detail. A service interface defines an interface to a service: what consumers need to know to determine if a service's capabilities meet their needs and if so, how to use the service. A ServiceInterface also defines as what providers need to know in order to implement the service.

B.1.3.4 Invoicing

Figure B.12 identified an InvoicingService capable of calculating the initial price for a purchase order, and then refining this price once the shipping information is known. The total price of the order depends on where the products are produced and from where they are shipped. The initial price calculation may be used to verify the customer has sufficient credit or still wants to purchase the products.

Figure B.13 shows a ServiceInterface that defines invoicing services. This ServiceInterface provides the Invoicing Interface and requires the InvoiceProcessing Interface.

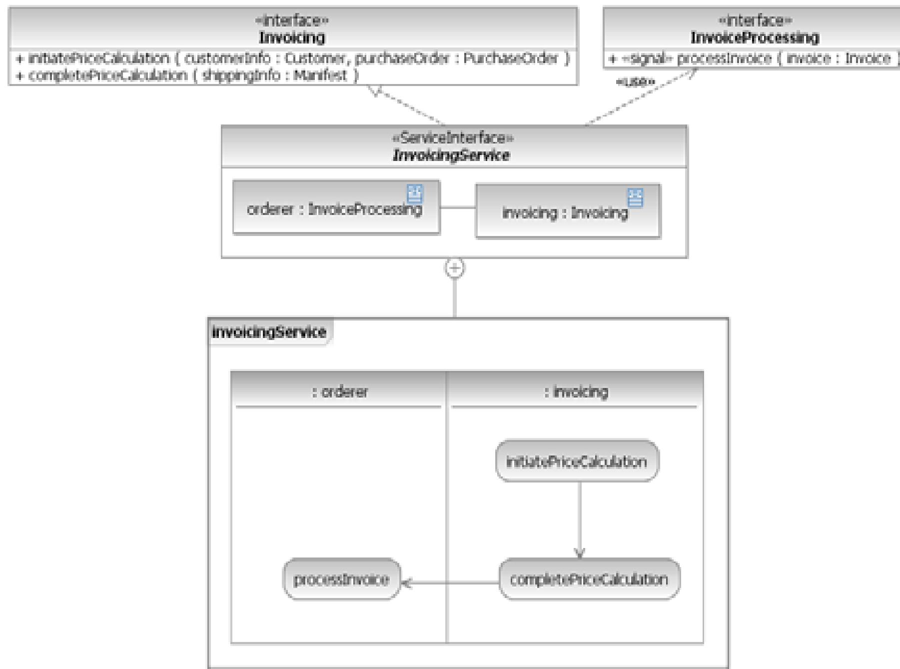


Figure B.13 - InvoicingService Service Interface

The protocol for the InvoicingService indicates that a consumer playing the role of an orderer must initiate a price calculation before attempting to get the complete price calculation. The orderer must then be prepared to respond to a request to process the final invoice. Some consumer requesting the invoicing service could do more than these three actions, but the sequencing of these specific actions is constrained by the protocol that is consistent with the behavioral part of the Process Purchase Order collaboration.

B.1.3.5 Production Scheduling

A scheduling service provides the ability to determine where goods will be produced and when. This information can be used to create a shipping schedule used in processing purchase orders.

The service interface for purchasing is sufficiently simple that it can be modeled as a simple interface. Only one interface is provided, none is required, and there is no protocol for using the service.

Figure B.14 - The Scheduling Service Interface

B.1.3.6 Shipping

A shipping service provides the capability to ship goods to a customer for a filled order. When the order is fulfilled, a shipping schedule is sent back to the client.

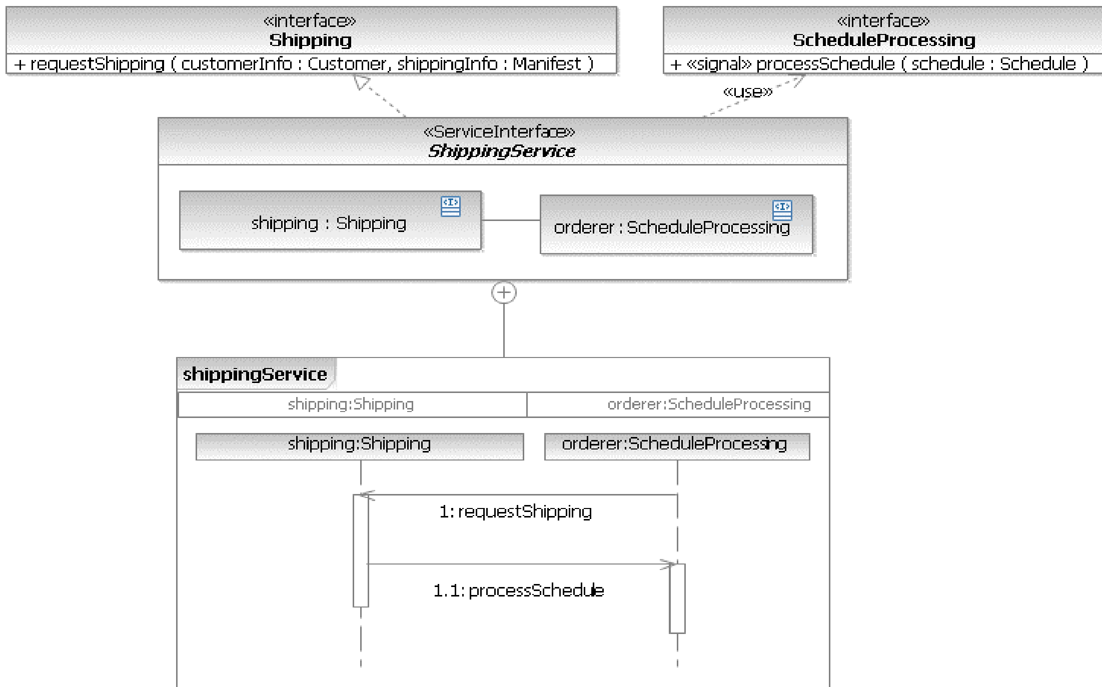


Figure B.15 - The Scheduling Service Interface

B.1.3.7 Shipping

A shipping service provides the capability to ship goods to a customer for a filled order. When the order is fulfilled, a shipping schedule is sent back to the client.

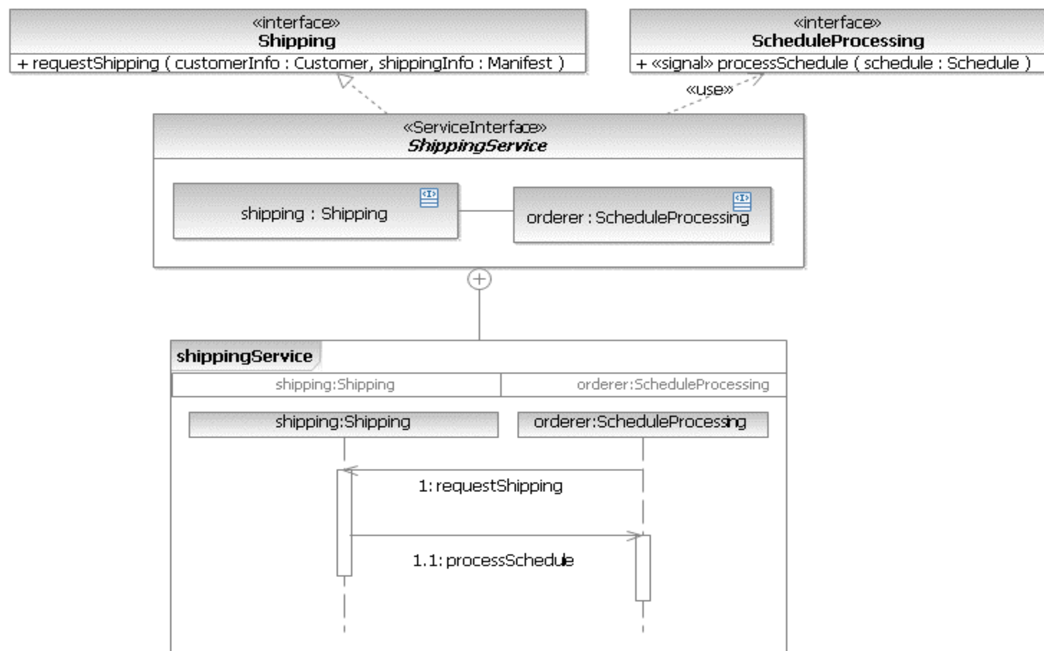


Figure B.16 - ShippingService Service Interface

B.1.3.8 Purchasing

The requirement was to create a new purchasing service that uses the invoicing, productions, and shipping services above according to the Process Purchase Order process. This will provide an implementation of the business process as choreography of a set of interacting service providers. Since this is such a simple service, no contract is required, and the service interface is a simple interface providing a single capability as shown in Figure B.17.

Figure B.17 - Purchasing Service Interface

Now that all the service interfaces have been specified the next step is to realize the services by creating participants that provide and consume services defined by these service interfaces.

Service Realization

Part of architecting a SOA solution is to determine what participants that will provide and consume what services, independent of how they do so. Particular participants may then be elaborated to show how they do so using the services of other participants or their own business process. These consumers and providers must conform to any fulfilled contracts as well as the protocols defined by the service interfaces they provide or require.

Each service offered by a participant must be implemented somehow. Each function (operation) will have a method (behavior) whose specification is the provided service operation. The design details of the service method can be specified using any Behavior: an Interaction, a\Activity, StateMachine, or OpaqueBehavior. Often a service participant's internal structure consists of an assembly of parts representing other service providers, and the service methods will be implemented using their provided capabilities.

B.1.3.9 Invoicing

The Invoicer service Participant shown in Figure B.18 provides an invoicing Service defined by ServiceInterface InvoicingService. That is, the invoicing Service provides the Invoicing interface, requires the InvoiceProcessing interface, and the design of the service operations must be compatible with the protocol for the service interface. The invoicing Service can also specify the possible bindings provided by the Invoicer Participant for use in connecting with other service participants.

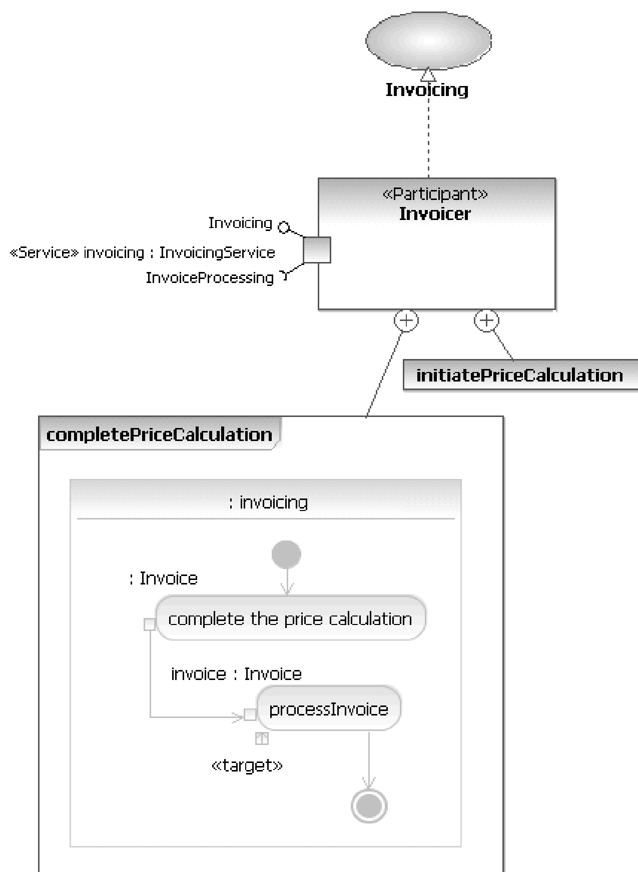


Figure B.18 - Invoicer Service Provider providing the invoicing service

The Invoicer service provider realizes the Invoicing use case (details not shown) that provides a means of modeling its functional and nonfunctional requirements.

Figure B.18 also shows the design of the implementation of the initiatePriceCalculation and completePriceCalculation service operations. The design of the initiatePriceCalculation is modeled as an OpaqueBehavior whose specification is the initiatePriceCalculation operation provided by the invoicing services. The design of the completePriceCalculation operation is also shown in Figure B.18. As you can see, this design is consistent with the ServiceInterface protocol since the processInvoice operation is invoked on the invoicing service port after the price calculation has been completed.

B.1.3.10 Production Scheduling

The Productions component shown in Figure B.19 provides a scheduling service defined by the Scheduling service port interface (in this case the type of the Service simple Interface).

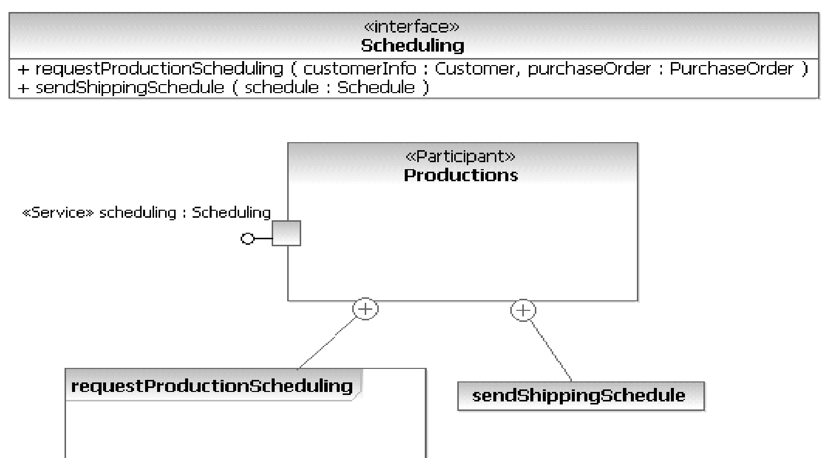


Figure B.19 - The Productions Service Provider

B.1.3.11 Shipping

A Shipper specification shown in Figure B.20 specifies a service provider that provides a shipping service defined by the Shipping service port interface. This specification component is not a provider of the shipping service. Rather it defines a specification for how to ship goods to a customer for a filled order that can be realized by possibly many different designs over time.

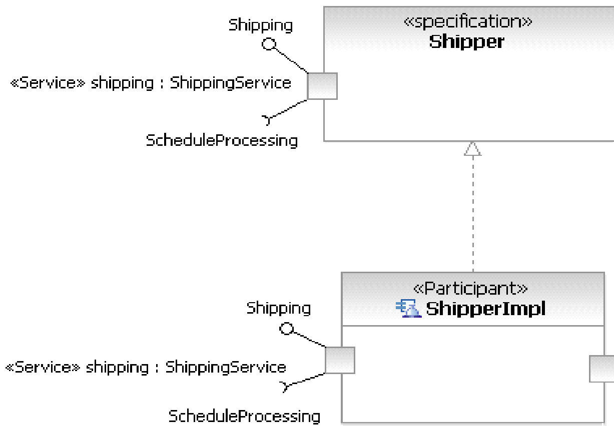


Figure B.20 - The Shipper Service Provider

The provider component ShipperImpl represents one such design that realizes the Shipper specification. This realization must provide and require all the services of all specifications it realizes, but may provide or use more depending on its particular design. Specifications therefore isolate consumers from particular provider designs. Any realization of the Shipper specification can be substituted for a reference to Shipper without affecting any connected consumer.

B.1.3.12 Purchasing

The purchase order processing services are specified by the Purchasing interface, and provided by the OrderProcessor provider as shown in Figure B.21. This participant provides the Purchasing Service through its purchasing port.

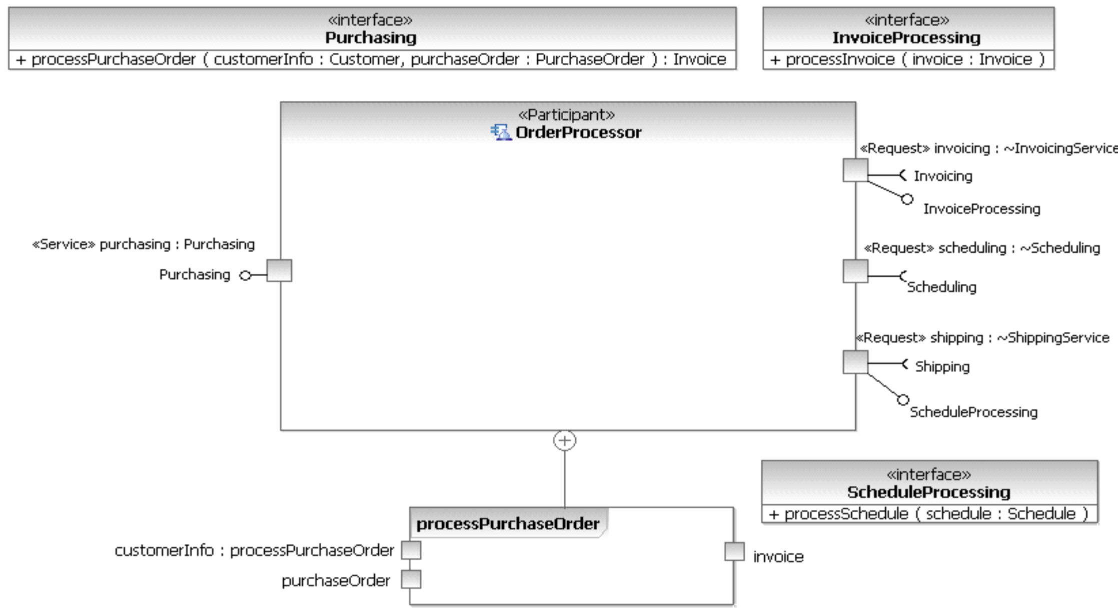


Figure B.21 - The OrderProcessor Service Provider

The OrderProcessor Participant also has Requisitions to for three Services: invoicing, scheduling and shipping. These providers of these services are used by the OrderProcessor component in order to implement its Services.

This example uses an Activity to model the design of the provided processPurchaseOrder service operation. The details for how this is done are shown in the internal structure of the OrderProcessor component providing the service as shown in Figure B.22.

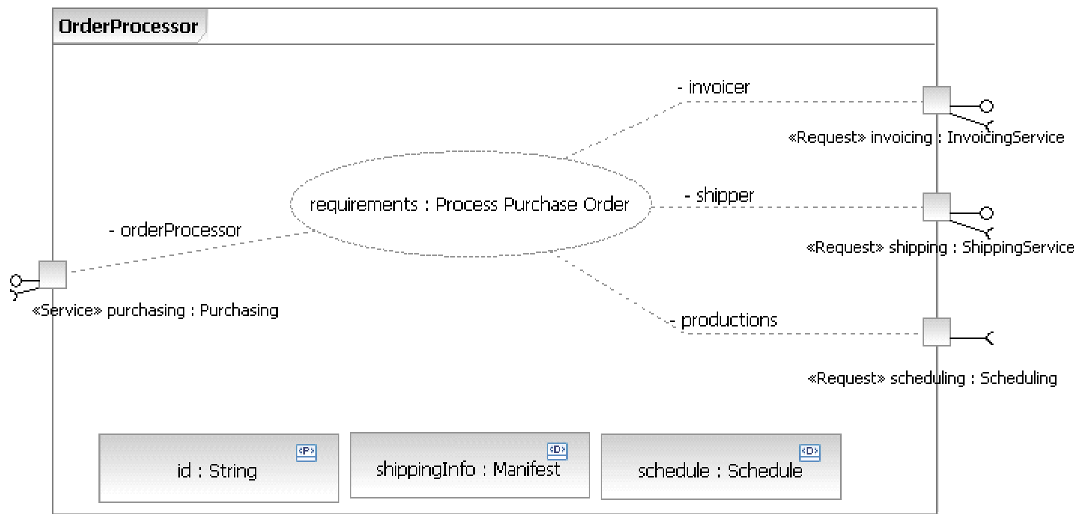


Figure B.22 - The Internal Structure of the OrderProcessor Service Provider

The internal structure of the OrderProcessor component is quite simple. It consists of the service ports for the provided and required services plus a number of other properties that maintain the state of the service provider. The id property is used to identify instances of this service provider. This property may be used to correlate consumer and provider interaction at runtime. The schedule and shippingInfo properties are information used in the design of the processPurchaseOrder service operation.

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

1. an ownedBehavior (Activity, Interaction, StateMachine, or OpaqueBehavior) that is the method of the service Operation, or

2. an AcceptEventAction (for asynchronous calls) or AcceptCallAction (for synchronous request/reply calls) in some Activity belonging to the component. This allows a single Activity to have more than one (generally) concurrent entry point controlling when the provider is able to respond to an event or service invocation. These AcceptEventActions are usually used to handle callbacks for returning information from other asynchronous CallOperationActions.

The OrderProcessor component has an example of both styles of service realization as shown in Figure B.23. The processPurchaseOrder operation is the specification of the processPurchaseOrder Activity which is an owned behavior of OrderProcessor.

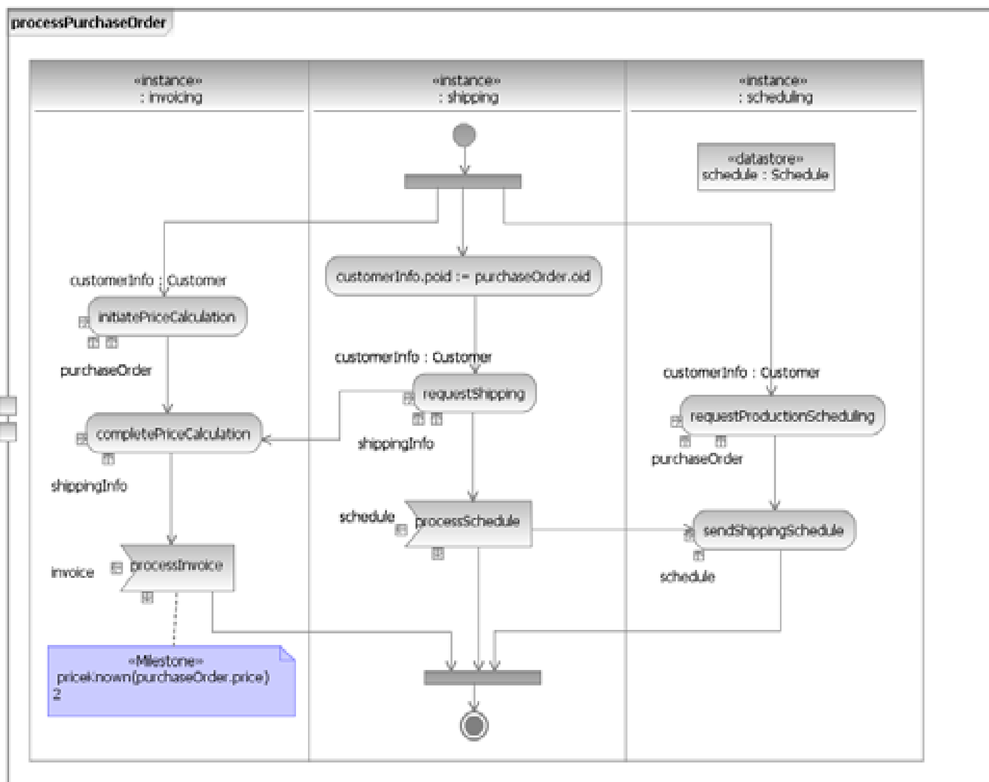


Figure B.23 - The processPurchaseOrder Service Operation Design

This diagram corresponds very closely to the BPMN diagram and BPEL process for the same behavior. The InvoiceProcessing and ShippingProcessing service operations are realized through the processInvoice and processSchedule accept event actions in the process. The corresponding operations in the interfaces are denoted as “trigger” operations to indicate the ability to respond to AcceptCallActions (similar to receptions and AcceptEventActions where the trigger is a SignalEvent).

Fulfilling Requirements

The OrderProcessor component is now complete. But there are two things left to do. First the OrderProcessor service provider needs to indicate that it fulfills the requirements specified in the collaboration shown in Figure B.11. Second, a Participant must be created that connects service providers capable of providing the OrderProcessor's required services to the appropriate services. This will result in a deployable Participant that is capable of executing. This section will deal with linking the SOA solution back to the business requirements. The next section covers the deployable subsystem.

Figure B.11 describes the requirements for the OrderProcessor service provider using a Collaboration. A CollaborationUse is added to the OrderProcessor service Participant to indicate the service contracts it fulfills as shown in Figure B.24.

The CollaborationUse, called requirements, is an instance of the Purchase Order Process Collaboration. This specifies the OrderProcessor service provider fulfills the Purchase Order Process requirements. The role bindings indicate which role the parts of the service Participant plays in the collaboration. For example, the invoicing requisition plays the invoicing role. The purchasing service plays the orderProcessor role.

Assembling Services

The OrderProcessor, Invoicer, Productions and Shipper Participants are classifiers that define the services consumed and provided by those participants and how they are used and implemented. In order to use the providers, it is necessary to assembly instances of them in some context, and connect the consumer requisitions to the provider services through service channels.

The Manufacturer Participant shown in Figure B.24 represents a complete component that connects the OrderProcessor service provider with other service providers that provide its required services.

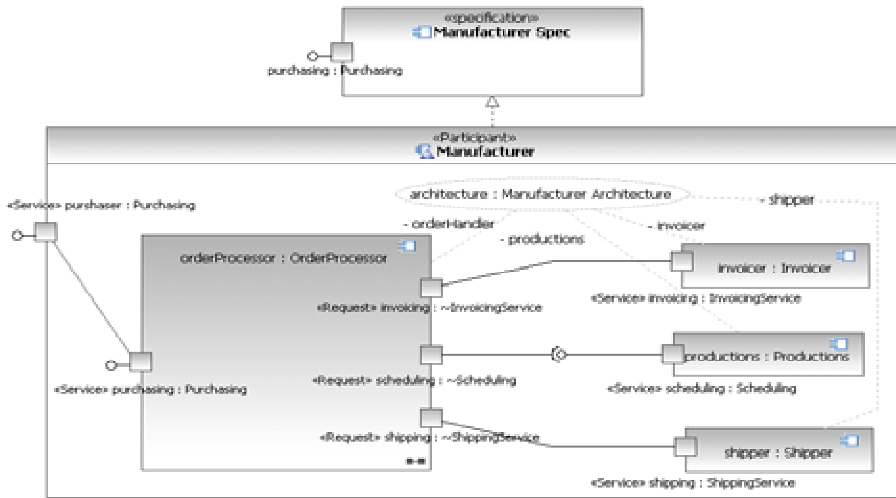


Figure B.24 - Assembling the Parts into a Deployable Subsystem

Figure B.24 also shows how the Manufacturer participant provides the purchaser service by delegating to the purchasing service of the Order processor.

The Manufacturer Participant is now complete and ready to be deployed. It has specific instances of all required service providers necessary to fully implement the processPurchaseOrder service. Once deployed, other service consumers can bind to the order processor component and invoke the service operation.

Services Data Model

The Customer Relationship Management service data model defined in package org::crm defines all the information used by all service operations in the PurchaseOrderProcess model in this example. This is the information exchanged between service consumers and providers. Messages are used to indicate the information can be exchanged without regard for where it is located or how it might be persisted.

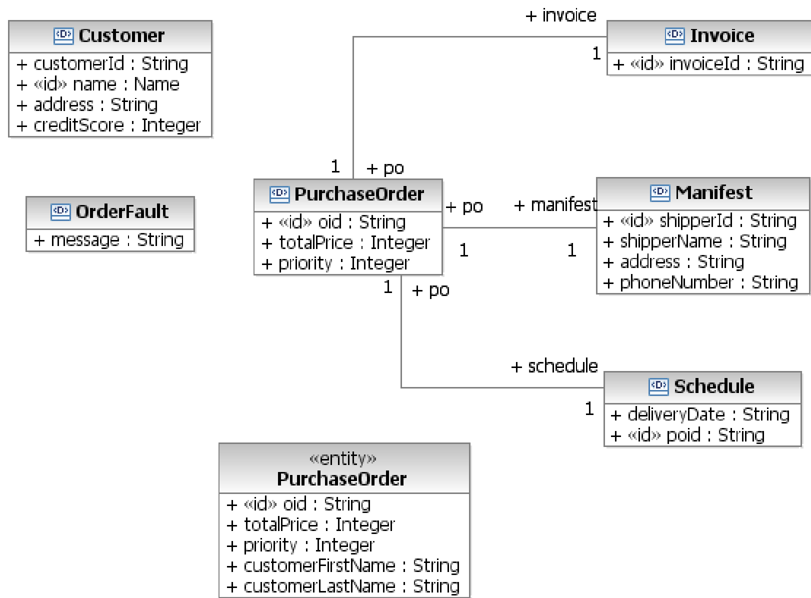


Figure B.25 - The CRM Data Model

Annex C: Purchase Order Example with Fujitsu SDAS/SOA

(informative)

This informative annex presents an example of service discovery using the KANAME Entity approach (http://www.fujitsu.com/downloads/MAG/vol42-3/paper05.pdf). It provides an example of a services model to establish a foundation for understanding the specification details. Service modeling involves many aspects of the solution development lifecycle. It is difficult to understand these different aspects when they are taken out of context and explained in detail. This example will provide the overall context to be used throughout the specification. It grounds the concepts in reality, and shows the relationships between the parts. The example is elaborated in other sections in order to explain specification details. The stereotypes used describe the minimum notation extensions necessary to support services modeling. These stereotypes may be viewed as either keywords designating the notation for the SoaML metamodel elements, or stereotypes defined in the equivalent SoaML profile.

C.1 Introduction

This material provides purchase order example, which is given in the UPMS RFP (soa/2006-09-09) to shows concept of Fujitsu SDAS/SOA. The Fujitsu SDAS/SOA prescribes to specify class diagram, state machine, and “Service Architecture” diagram for the application. In this material, these diagrams are specified.

There are some premises for this specification:

- Determine the producing factories to make delivery cost lower considering the productivity of the factories.
- The delivery plans are determined on the morning of the delivery day, (that is, delivery plans cannot be updated on the delivery day.) The delivery plans can be updated till the delivery day.
- Draft cost doesn’t include the delivery cost, that is, draft cost can be calculated as price * amount.
- A Detail_Invoice is always issued for each Purchase order slip, that is, there are no production defects and order changes.
- Detail_Invoices are made up to an Invoice at the end of the month.

C.2 Each specification model

In this application, KANAME of KANAME entity is a “PurchaseOrder,” because “PurchaseOrder” control entire behavior and play a main role. And, considering related entities, all other KANAME entities can be extracted. SoaML example of Class diagram is used.

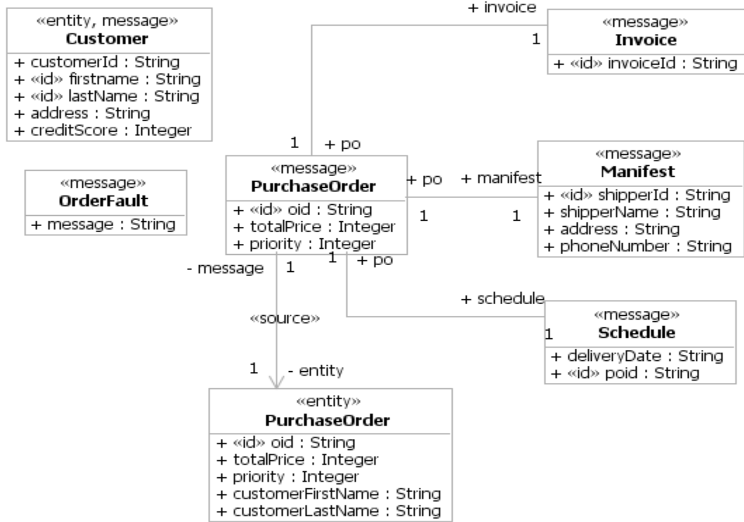


Figure C.1 - Entity Model (Class diagram)

C.2.1 StateMachine

Based on KANAME entity model, state machine for each KANAME entity is described and all state machines are merged into only one state machine that is allied with each other coordinating its interaction points. Then, we can get the following state machine.

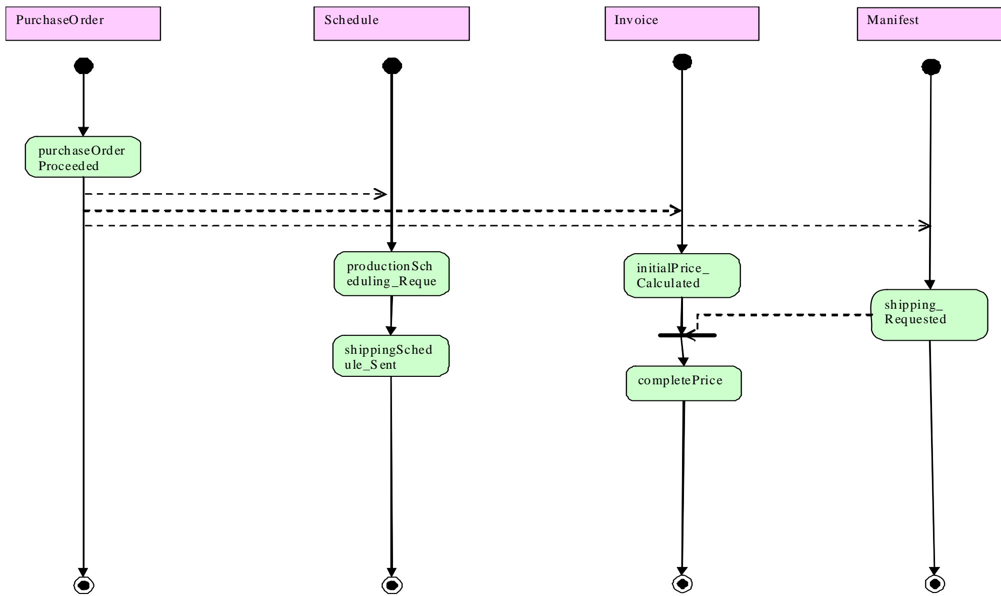


Figure C.2 - State Machine

Each state transition is caused by each operation. Then, such each operation is defined as required "Service." Besides, considering roles of services, those services are grouped into same role. Then, we can get Service Architecture diagram deploying such service.

Annex D: BPDM Mapping

(informative)

SoaML is intended to work in concert with a business process specification such as is specified by BPDM and eventually BPMN-2. The business process specification may be "larger" than the Services Architecture and thus show how services are provided and used within a process, it may be "smaller" than the Services Architecture in that it may be the process of a particular Participant or a business process may be the process view of Services Architecture. In that BPDM encompasses concepts of SOA as well as process-activity modeling it covers more than SoaML.

The best practice connection is that there is a business process view of each Services Architecture where the roles in the Services Architecture map to the roles (or lanes) in the business process.

In each case the concepts of the SOA profile map well to the process view as shown in the following informal table:

SoaML Concept	BPDM Concept
Interface & Service Interface	Interaction Protocol
Message	Type
Participant	Processor Role
Role in service contract	Interaction Role
Service Contract	Interaction Protocol
Service Interface	Implied by interaction protocol
Service Port	Involved interaction association
Service Realization	Processor Role
Service Contract Use	Interaction
Services Architecture	Process