Service oriented architecture Modeling Language (SoaML) - Specification for the UML Profile and Metamodel for Services (UPMS)

Revised Submission
OMG document: ad/2008-08-04

Submitters

Adaptive
Capgemini
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
Visumpoint
MID GmbH
NKUA – University of Athens
Oslo Software
SINTEF
THALES Group
University of Augsburg
Wilton Consulting Group

Primary Contact:

Arne J. Berre, SINTEF
email: Arne.J.Berre@sintef.no

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

DISCLAIMER OF WARRANTY

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

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).

Revision History

Date	Version	Description	Author
December 2006	0.01	Initial version submitted for review to SoaML submitters	Jim Amsden
May 2007	1.0	Initial version from submission team.	Jim Amsden
December 2007	1.1	Revised submission	Jim Amsden, Jim Odell
December 2007	1.2	Revised submission	Jim Amsden, Jim Odell
March 2008	1.3	Merged submission with SOA Profile	Jim Amsden
May 28th 2008	1.4	Revised joint submission UPMS/SoaML	Arne J. Berre
August 25th 2008	1.8	Revised joint submission SoaML for UPMS	Arne J. Berre

Table of Contents

PART I

Preface

About this Specification

Overview of this Specification

The SoaML (Service oriented architecture Modeling Language) specification is created in response to the UPMS (UML Profile and Metamodel for Services) RFP and describes a UML profile and metamodel for the design of services within a service-oriented architecture.

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

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

The SoaML metamodel is based on the UML 2.0 metamodel L2 and provides minimal extensions to UML, only where absolutely necessary to accomplish the goals and requirements of service modeling. The specification takes advantage of the package merge feature of UML 2.0 to merge the extensions into UML. The profile provides a UML specific version of the metamodel that can be incorporated into standard UML modeling tools.

Intended Audience

This document is intended for architects, analysts and designers who will be specifying service oriented solutions. It assumes a knowledge of UML 2.0.

Organization of this Specification

The specification includes the following parts and chapters:

Part I

- Non-normative introduction to the SoaML specification
- Resolution of UPMS RFP Mandatory and Optional Requirements

Part II

- Scope: specifies the scope of services modeling covered.
- Compliance: compliance with SoaML requires that the subset of UML required for SoaML is implemented. This specifies the optional services modeling features and their compliance levels.
- Normative References: References to other adopted specifications.
- Terms and Definitions: Formal definitions that are taken from other documents.
- Symbols: Identification and definition of special symbols required to read this specification.
- SoaML UML Profile Specification: Desribes the SoaML UML profile
- Classification: Describes the adaptation of RAS to model classifiction
- BMM Integration: Describes the integration with the Business Motivation Metamodel
- SoaML Metamodel: Describes the SoaML Metamodel
- Non-normative Appendices:
- A: Sample Web Services Output: This will be added later
- B: Conformance to OASIS SOA Reference Model and others: Compare SoaML concepts with various SOA reference models and ontologies
- C and D: Example: Illustrates use of SoaML in practice
- E: SOA Patterns and Best practices: Describes useful non-normative SOA patterns and best practices.

Part III

- Changes to Adopted Specifications; recommends modification or retirement of adopted OMG specifications. (where none are recommended)

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: SoaML metaclasses, stereotypes and other 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*.

Additional Information

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 refered 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 is intended to be read in a non-sequential manner. Consequently, extensive cross-references are provided to facilitate browsing and search.

Acknowledgements

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

Submitters
Adaptive
Capgemini
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, the SoaML Submission Team would like to acknowledge the participation and contribution from the following individuals: Jim Amsden, Irv Badr, Bernhard Bauer, Arne Berre, Mariano Belaunde, John Butler, Cory Casanave, Bob Covington, Fred Cummins, Philippe Desfray, Andreas Ditze, 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, Dimirtu Roman, Mike Rosen, Stephen Roser, and Omair Shafiq, and Ed Seidewitz, Bran Selic, Kenn Hussey, Fred Mervine.

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

As noted, OMG specifications address middleware, modeling, and vertical domain frameworks. A catalog of all OMG Specifications Catalog is available from the OMG website at *http://www.omg.org/technology/documents/spec_catalog.htm*.

Specifications within the Catalog are organized by the following categories:

OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

Platform Specific Model and 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
250 First Avenue
Suite 100
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*

Resolution of RFP Mandatory and Optional Requirements

Mandatory Requirements

6.5.1 Required Profile

6.5.1.1 Submissions to this RFP shall provide a MOF metamodel and equivalent Profile extending UML for Services Modeling as specified in other mandatory requirements specified in this RFP.

The submission is a MOF metamodel that imports UML2 compliance level L3 to define a new capability that can be merged with L3 and above to extend UML2 with service modeling capability. An equivalent profile is provided that use stereotypes with the same names and properties as the new metaclasses, and extend either the merged metaclass, or metaclass superclass as defined in the services metamodel.

6.5.2 UML Compatibility

6.5.2.1 These metamodel and profile extensions can extend, but not conflict with existing UML semantics for extended metaclasses.

The submission metamodel uses package merge and generalization/specialization to ensure that the services metamodel only extends UML2. It does not violate any package merge rules, or substitutability rules for specialization/generalization.

6.5.3 Notation

6.5.3.1 Submissions shall specify icons for stereotype extensions to UML in order to extend the UML notation for services modeling.

The submission provides notation options for new UML keywords to designate new metaclasses. The keywords are designed to be the same as the stereotype display names in the profile. There is also a notation option providing new icons or shapes for some new metaclasses. The icons are the same between the metaclass and the profile. Every attempt has been made to keep the notation extensions consistent with UML2 practice. This includes avoiding depending on color and keeping shapes easy to draw by hand.

6.5.4 Platform Independent

6.5.4.1 Submissions shall express the intent of services models rather than any specific means by which that intent may be realized by some runtime platform.

The SoaML submission models services as an extension to UML2 Component models. Participants extend Components with the ability to have Services and Request which are kinds of Port. Service capability implementations may be modeled using UML2 Behaviors. There is no attempt to use SoaML to model specific platforms such as JEE or

Web Services. Participants do introduce the ability to have Bindings which describe additional characteristics about the Connectors between services and requisitions. These are intended to capture characteristics about the connections that may effect services architecture and design decisions such as distribution and possible communication protocols.

6.5.4.2 Submissions shall include a non-normative mapping from the Services Profile to Web Services (XSD, WSDL, BPEL, etc.) in order to demonstrate the completeness and applicability of the submission.

Appendices will be provided during the finalization phase of the standard to include a non-normative mapping from the Services Profile to Web Services (XSD, WSDL, BPEL, etc.).

6.5.5 Specification of Service Contracts

6.5.5.1 Submissions shall provide a means of specifying contracts (i.e., requirements) that must be met by service specifications or realizations. Service contracts shall include:

- *a. The ability to specify bi-directional, complex, long-lived and asynchronous interactions between service consumers and providers*
- *b. The functions specified by the contract.*
- *c. The ability to realize UseCases that capture contract requirements.*
- *d. The participants in the contract and the roles they play.*
- *e. The responsibilities of those roles in the context of the contract. It shall be possible to specify these responsibilities using service interfaces, service specifications, or particular service providers (see Section 6.5.13).*
- *f. Connectors indicating possible interactions between roles.*
- *g. Behavioral rules for how the roles interact.*
- *h. Constraints or objectives that must be met.*

The submission provides a new metaclass called ServiceContract that extends UML2 Collaboration. Collaboration has all the capabilities specified above and specifies them in an architecturally neutral way. That is, Collaborations say what systems have to do, not how they do it (how, or the ways to do it are described through ownedbehaviors). CollaborationUse is used to bind the parts of the solution to the roles they play in a ServiceContract (who is taking part in the contract). Collaboration and CollaborationUse are extended with property isStrict. This property indicates whether the role bindings must be type compatible or not. In summary:

a. The use of UML Collaboration as the basis for contract, ensures the ability to specify bi-directional, complex, long-lived and asynchronous interactions between service consumers and providers. Who are playing the role of provider or consumer is defined during the fulfillment.
b. The functions are specified in the Interfaces, ServiceInterfaces or Participants linked to the collaboration parts.
c. Collaboration is a usual mean to realize use cases, as the contract inherits from collaboration it also inherits that ability

d. Through parts in the contract we specify the interfaces that specify the roles, and in the fulfillment phase we can establish a role-binding between the part and any participant implementing the interface.
e. The responsibilities in the contract can be specified using Interfaces, ServiceInterfaces or Participant specifications binding them to the parts in the collaboration.
f. Regular UML2 Connector can be use to indicate possible interactions between roles
g. This is implemented by the ownedbehaviors of the collaboration that the contract extends.
h. UML2 Constraints may be used to model objectives and rules for a ServiceContract.

6.5.5.2 Service contracts shall be consistent with the upcoming BPDM choreography and collaborations.

Appendices will be provided during the finalization phase of the standard to address the consistency with BPDM.

6.5.6 Service Contract Fulfillment

6.5.6.1 Service specifications and/or providers shall have a means to indicate the service contracts they fulfill and how they provide for the roles in the contract.

CollaborationUse in SoaML specializes UML2 CollaborationUse to indicate the fulfillment of a ServiceContract. Fulfillments can be used in any Classifier including other ServiceContracts to show ServiceContracts of roles engaged in other ServiceContracts. ServiceInterfaces may fulfill contracts as can specification Participants, Participants and Agents. These CollaborationUse constrain the classifiers that contain them.

6.5.6.2 Submissions shall specify a means for supporting both strict and loose contract fulfillment.

a. Strict contract fulfillment shall mean that a service specification or service provider must be conformant with the service contracts it fulfills. Submissions shall provide a complete definition of conformance which includes, but is not limited to type, behavior, and constraint conformance with the constraints, responsibilities and rules for the role in the contract.

b. Loose contract fulfillment shall mean the modeler warrants the service consumers and providers are capable of playing the indicated roles, but this is not validated by the metamodel, nor do all roles in the contract have to be filled by the service specification or provider.

SoaML ServiceContract and CollaborationUse have Boolean Property isStrict which is used for this purpose. If the ServiceContract has isStrict=true, then all CollaborationUse of that Contract must also have isStrict=true, and all parts bound to roles in the ServiceContract through CollaborationUses must be type compatible with the role they play. If ServiceContract has isStrict=false, then individual CollaborationUses may have

isStrict either true or false indicating whether that particular CollaborationUse is intended to be strictly followed.

6.5.6.3 The use of service contracts in any services model shall be optional. Services contracts also need not be fulfilled by any service provider (although tools would be encouraged to report on such contracts).

SoaML does not require ServiceContracts or CollaborationUses. Their use is completely up to the modeler. ServiceContracts may be very useful for capturing semantically rich requirements or Service Level Agreements in ServiceInterfaces, but they are not required. ServiceContracts may be specified as a means for documenting service requirements, but need not be explicitly fulfilled. This allows SoaML to be flexible in support for many service oriented methodologies.

6.5.7 Service Specification

6.5.7.1 Submissions shall provide a means of specifying service specifications independently of how those services are provided or implemented. Such service specifications shall include:

a. *Specification of possibly many Service Interfaces where all operations are considered available in distributed, concurrent systems with no assumptions about global synchronization, control or shared address spaces. (Service interface definition in Glossary B.2)*
b. *Service Operations in Service Interfaces*
c. *Operation pre and post conditions, parameters and exceptions*
d. *Constraints service providers are expected to honor*
e. *Service interaction points through which service interfaces are provided and required in order to decouple consumers and providers*
f. *Behaviors as methods of operations provided by the Service Specification indicating the behavioral semantics that must be supported by any realizing service provider.*

The submission provides a new metaclass called ServiceInterface. ServiceInterfaces allows specifying services in SoaML. ServiceInterfaces can specify the use and provision of operations between the service consumer and the service provider. This is done through the realization and usage of regular UML Interfaces by the ServiceInterface.

a. Many ServiceInterfaces can be used during the design of a system to specify a set of services interacting in a distributed, concurrent environment.
b. The operations of the ServiceInterfaces are specified in their linked Interfaces
c. Operations pre and post conditions, parameters and exceptions can be modeled using UML2 elements.
d. Constrains for the operations can be described using UML2 elements.
e. Participants can have Service and Request ports through which ServiceInterfaces can be provided and required.
f. ServiceInterfaces extends the UML2 Class element, as such, it can contain one or more owned behaviors. This can be used to explain the way in which the different operations of the serviceInterfaces are provided and required to achieve certain goals.

6.5.7.2 The use of service specifications in any services model shall be optional and are provided for additional decoupling between service consumers and service providers. Developers decide the degree of coupling that is tolerable for their solutions.

As in UML2, specification Components, Participants and Agents are optional in SoaML. Specification components may be used as types for parts in component assemblies, a component's internal structure. These assemblies are considered incomplete until some realizing participant is chosen to actually consume and provide services. How and when this is done is not specified in SoaML. It could be done at modeling time by referencing particular realizing components, at deployment time, or dynamically at runtime.

6.5.8 Specification of Service Data

6.5.8.1 Submissions shall provide a means of indicating structured service data or information exchanged between service consumers and services providers.

The SoaML provides the DataMessage element to explicitly indicate structured service data or information exchanged between service consumers and service providers. The internal structure of the DataMessage can be described using UML2 elements such as classes and associations.

6.5.8.2 Submissions shall provide a means of indicating components of service data representing attachments containing opaque information exchanged between consumers and providers. Such attachments shall provide for an indication of the type of the opaque data such as MIME type where applicable, or may be represented by extensions to UML primitive types.

The SoaML provides the Attachment element to represent information elements that have their own identity and which are separable form the main message without losing their meaning.

6.5.8.3 The usage semantics of service data shall make no assumptions with regard to global synchronization, control or shared address spaces.

The SoaML provides the Attachment element to represent information elements that have their own identity and which are separable form the main message without losing their meaning.

6.5.9 Synchronous and Asynchronous Service Invocations

6.5.9.1 Submissions shall support synchronous and asynchronous service invocation.

SoaML does not provide specific elements or attributes to distinguish synchronous or asynchronous service invocations, beyond those that already exists in UML. Both can be used in the implementation of the services specified with the SoaML.

6.5.9.2 Synchronicity shall be a property of the invocation of a service operation, not its specification in a service interface or its implementation in a service provider. That is, a service provider may expect a service to be invoked a certain way, but it is up to clients using the service to determine how the service is to be used. In particular, a service that has output or return parameters may be called synchronously or asynchronously. It is up to the client invoking the service to determine if the output is needed or not.

SoaML leaves the decision on the way in which the operations are invoked to the implementation of the clients and the capabilities of the implementation technologies. It is up to the implementer and the capabilities of the technologies it use to decide if the operations are invoked synchronously or asynchronously.

6.5.10 Service Event Handling and Generation

6.5.10.1 Submissions shall provide a means of designating the ability of a service specification or provider to receive an event.

Events are not addressed in this submission, beyond what is already supported in UML. A new OMG RFP on an Event Metamodel and Profile (EMP) is in preparation to deal with more complex Event modeling,

6.5.10.2 Submissions shall provide a means of generating events targeted at a specific service provider or broadcast to interested providers.

See comment under 6.5.10.1.

6.5.10.3 Service operations that respond to events shall not have outputs, are always invoked asynchronously, and may raise exceptions.

See comment under 6.5.10.1.

6.5.11 Service Parameters

6.5.11.1 All parameter types of all operations provided by a service specification or service provider shall be either primitive types or service data in order to minimize coupling between service consumers and service providers.

The operations in SoaML are specified in the same way in which they are specified in UML2, therefore the can be parameterized with primitive types and structured types.

6.6.12 Service Consumers

6.5.12.1 Submissions shall provide a means of designating a consumer which requires services provided by other service providers.

Service consumers in SoaML are those participants that own Request ports. Request ports specify that the participant, who owns them, needs those services to achieve their objectives. The participant in that case, does not implement the services, it use them.

6.5.13 Service Providers

6.5.13.1 Submissions shall provide a means of designating a service provider.

Service Providers in SoaML are those participants that own Service ports. Service ports specify that the participant, who owns them, provides those services to achieve their objectives. The participant in that case, implements the services.

6.5.13.2 Service providers shall provide and require services only through service interaction points. Service providers shall not directly realize or use service interfaces other than realize service specifications. This ensures isolation and decoupling between consumers and providers.

ServicesInterfaces can only be provided through Service Ports, and can only be used through Request Ports. Service and Request ports are both interaction points and they ensure the isolation and decoupling between consumers and providers.

6.5.13.3 A service provider shall be able to realize zero or more service specifications.

Participants in SoaML can have several Service ports.

6.5.13.4 A service provider must be conformant to all service specifications it realizes. Submissions shall provide a complete definition of conformance which includes, but is not limited to type, behavior, and constraint conformance. Such conformance rules shall be consistent with the conformance rules for fulfilling service contracts.

SoaML provides the semantic implications of the realization and usage of ServiceInterfaces by service participants.

6.5.13.5 An interaction point of a service provider shall be able to provide and/or require at least one service interface.

Service and Request Ports can be typed with Interfaces and ServiceInterfaces. This allows to associate one or more Interfaces to an interaction point.

6.5.13.6 A service provider shall be able to specify binding information applicable to all its interaction points for use in connecting to service consumers.

SoaML does not cover the specification of binding information.

6.5.13.7 A service interaction point shall be able to restrict and/or extend the bindings available through that interaction point overriding the binding information of its containing service provider.

SoaML does not cover the specification of binding information.

6.5.14 Service Realizations

6.5.14.1 Submissions shall provide a means of specifying the realizations of provided service operations through owned behaviors of the service provider.

SoaML ServiceInterface extends UML2 Class element and it can have one or more ownedBehaviors. These Behaviors can be used to specify the way in which the operations are used to implement the capabilities of the ServiceInterface.

6.5.14.2 Submissions shall provide multiple styles for specifying behavior implementations including but not limited to UML activities, interactions, state machines, protocol state machines, and/or opaque behaviors. These may differ from the means used for the same operation in any realized service specification.

SoaML ServiceInterface Behaviors can specified with any of the UML behavior implementation mechanisms, Mechanisms such as UML activities, interactions, state machines, protocol state machines, and/or opaque behaviors.

6.5.15 Composing Services from other Services

6.5.15.1 Submissions shall specify how services are composed from other services.

The way in which services are composed form other services can be describe in the behavior of the Participants who provides and uses those services through its interaction points. It is also possible to specify the way in which the different services of the parts of a ServiceContract are composed. This is done in the behaviors of the ServiceContract.

6.5.14.2 Submissions shall make no assumptions about or constraints on the number of levels of composed services, or impose arbitrary distinctions between specific composition levels.

SoaML does not impose restrictions on the number of level of composed services.

6.5.16 Connecting Service Consumers with Service Providers

6.5.16.1 Submissions shall support service channels for connections between usages of service consumers and service providers in the internal structure of some containing element.

UML2 Connectors can be used to identify service channels in the internal structure of Participants, ServiceContracts, and ServiceArchitectures.

6.5.16.2 Submissions shall support different degrees of coupling between connected service consumers and providers by allowing service providers to be specified by any of the following:

a. *A service interface (supports minimal coupling by allowing connections to any service providing that service interface)*

b. A service specification (supports moderate coupling by allowing connections to any service provider realizing the service specification)

c. A service provider (provides direct coupling to a specific service provider)

SoaML Provides elements to allow the service based system modeling with different levels of coupling.

a. UML2 Interfaces can be used to identify connections to the ServiceInterface
b. SoaML ServiceInterface can be used to specify connections to any service Provider
c. SoaML Provider can be used to specify connections among different ServiceInterfces

6.5.16.3 Submissions shall provide a way for service channels to specify a binding from possible bindings specified on the connected interaction point of the service provider and bindings expected by the interaction point of the connected service consumer.

UML2 Connectors can be used to specify bindings between providers and consumers.

6.5.17 Customizing and Extending Services

6.5.17.1 Submissions shall specify a means of customizing and extending services that shall include, but not be limited to, the following facilities (as specified by UML):

a. Service specification or provider properties that can be configured by setting values during service development, or deferred to service deployment or runtime.

b. Refinement and redefinition through generalization/specialization.

c. Pattern or template specification and instantiation.

UML2 can be used to customize or extend elements modeled with SoaML.

6.5.18 Service Partitions

6.5.18.1 Submissions shall provide a means of organizing service specifications and/or providers into partitions for organization and management purposes.

UML2 packages can be used to organize specifications and/or participants for organization and management purposes.

6.15.18.2 Submissions shall provide a means of specifying constraints on the connections between service partitions.

UML2 constrains can be used in the connections between service partitions.

6.5.19 Service Model Interchange

6.5.19.1 The resulting metamodel and profile shall be made available as an XMI document based on the UML and MOF to XMI mapping specifications and the UML rules for exchanging profiles using XMI.

Appendices will be provided during the finalization phase of the standard to address the XMI document and mapping.

6.5.19.2 Instances of the service metamodel shall be exchanged using XMI as specified in the MOF to XMI mapping specification. Instances of services models created using the equivalent UML profile shall be exchanged using XMI as specified by the UML rules for exchanging instances of UML models with applied profiles. Submissions shall define interchange compliance levels for each for these XMI document formats.

Appendices will be provided during the finalization phase of the standard to address the compliance levels for each for these XMI document formats.

Optional Requirements

6.6.1 Proposals may provide additional mappings to existing platforms and languages for service specification and execution.

SoaML does not address this requirement in its current stage

6.6.2 Submissions may provide a means for specifying the preferred encoding for service data exchange in order to maximize interoperability between service consumers and providers.

SoaML does not address this requirement in its current stage

6.6.3 Submissions may provide a means for specifying a binding metamodel for capturing structured binding information rather than simple strings.

SoaML does not address this requirement in its current stage

Responses to RFP Issues to be Discussed

6.7.1 Relationship to existing UML metamodel

Proposals shall discuss the relationship of submission model elements and UML2 modeling to demonstrate consistency with the UML metamodel. In particular, the following areas shall be discussed:

- *Relationship to UML2 component modeling*
- *Relationship between service specifications and UML2 «specification» Component*
- *Clarification concerning options for where services choreographies/communication protocols could be captured:*
    - *In a delegate part of a component*
    - *In an owned behavior or a component*
    - *In a collaboration between parts of another assembly component*

    - *In the contract of a connector*
    - *In a port class (the type of a Port)*
    - *Using nested classes*
- *Additional constraints needed for effective services modeling*
- *Formalizing SOA modeling best practices*
- *Relationship between service contracts and UML2 Collaborations and CollaborationUses as indicators of contract fulfillment*
- *The relationship between services and UML2 Ports and classes used to type ports.*
- *The use of UML2 Behaviors (Activity, Interaction, StateMachine, ProtocolStateMachine, and/or OpaqueBehavior) in service specifications and realizations.*
- *Relationship between service data and possible domain data used in service implementations.*

SoaML Participant is a subclass of UML2 Component and as such may perform as a UML «*specification*» component. However, While UML2 Components merely talk about their required and provided interfaces which are derived from, e.g., the types of the component's ports, SoaML Participant makes the offers to the surroundings more explicit. A SoaML Participant has owned ports that are either SoaML Services or SoaML Request. These again will have provided and required interfaces by being subclasses of UML2 Port. The types of the SoaML services or requests must be SoaML ServiceInterfaces. SoaML ServiceInterface is a concept that makes it possible to type both ends of a connector between a Service and Request by the same type.

Communication protocols are modeled in SoaML through the behaviors of UML2 Collaborations, specialized as SoaML ServiceContracts. Such ServiceContracts are applied through SoaML Fulfillments that bind the roles of the contacts. SoaML CollaborationUse is a subclass of UML2 CollaborationUse. Binding follows the general semantics of UML2 collaborations.

SoaML makes use of UML2 behaviors in the normal UML2 way. In particular it is a challenge for tools to assess the consistency between the behaviors of the SoaML Participants and the corresponding behaviors of SoaML ServiceContracts whose roles are bound to these participants.

SOA modeling best practices are a wide area and one cannot expect this field can be covered only through the core SoaML concepts mentioned above. SoaML have therefore also made minimal extensions to relate effectively to a few related standards and SOA approaches. Typically the SoaML metamodel define a few concepts that serve as mediators between SoaML core and the related metamodels.

SoaML also supports SOA where the services may be initiated on either side and where all participants are considered independent. Typically such SOA approach can be found in the telecom and embedded systems where the services may never terminate. The agent-oriented approach is one way to accommodate for this kind of SOA, and SoaML

has included concepts Agent (as a special Participant) and ProgressLabel (to determine progress in services that may run indefinitely).

6.7.2 Consistency Checks and Model Validation

Submissions shall discuss how the specification supports automated consistency checks for model validation particularly between service contracts, service specifications, and service realizations.

SoaML consistency is the same as UML2 consistency. The main challenge of consistency is that of CollaborationUse meaning the behavioral consistency between the bound parts of an actual composite structure and the specification given by a ServiceContract. The UML2 semantics leaves the concrete interpretation of such consistency to the users/vendors.

6.7.3 Applicability to Enterprise Service Bus Runtime Architectures

Submission shall discuss how the specification relates to or supports usage of Enterprise Service Bus (ESB) architectures.

SoaML is well suited to apply the Enterprise Service Bus Runtime Architectures since ESB is an attractive realization of the SoaML ServiceInterface concept. Furthermore the asynchronous messaging techniques advocated by ESB conform well with the SOA approach characterized by independent Agents.

6.7.4 Relationship to UDDI.

Submissions shall discuss how instances of services model may be used to capture information necessary to populate UDDI repositories with information necessary for service discovery and use.

While SoaML is mainly targeting the detailed technical specification for executable services, UDDI targets the metadata needed to find and bind services. The SoaML and the UDDI meet basically in the description of *tModels*, the UDDI detailed description of services. The SoaML Contracts may be coded into *bindingTemplates* and *tModels*.

PART II

Scope

The SoaML (Service oriented architecture Modeling Language) 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 Service)s 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.
- The current 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 the upcoming 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.

Compliance

Compliance with SoaML requires that the subset of UML required for SoaML is implemented, and the extensions to the UML subset required for SoaML are implemented as described in this submission. In order to fully comply with SoaML, a tool must implement both the concrete syntax (notation) and abstract syntax (metamodel) for the required UML subset and the SoaML extensions. Tools may implement SoaML using either the profile or by extending UML using package merge and the SoaML metamodel. In any case, tools are expected to be able to interchange XMI for both the UML profile and the metamodel extensions for the supported compliance level.

Tools that implement the profile may conform to the compliance levels described here by including only those stereotypes that correspond to the meta-classes defined in the compliance level.

Compliance Levels for SoaML

The following levels elaborate the definition of compliance for SoaML.

- **Level 1 (SoaML:L1) -** This level provides the core UML concepts contained in the UML L2 package and adds the capabilities provided by the Service package (Figure 1).

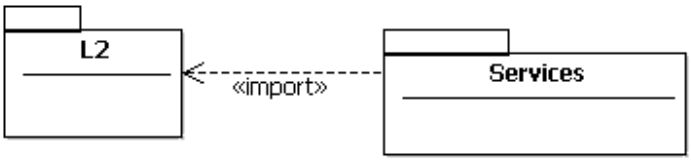

Figure 1: SoaML Compliance level SoaML:L1

- **Level 2 (SoaML:L2) -** This level extends the SoaML provided in Level 1 (SoaMLL1) and adds in integration with the OMG Business Motivation Model (BMM) (Figure 2).

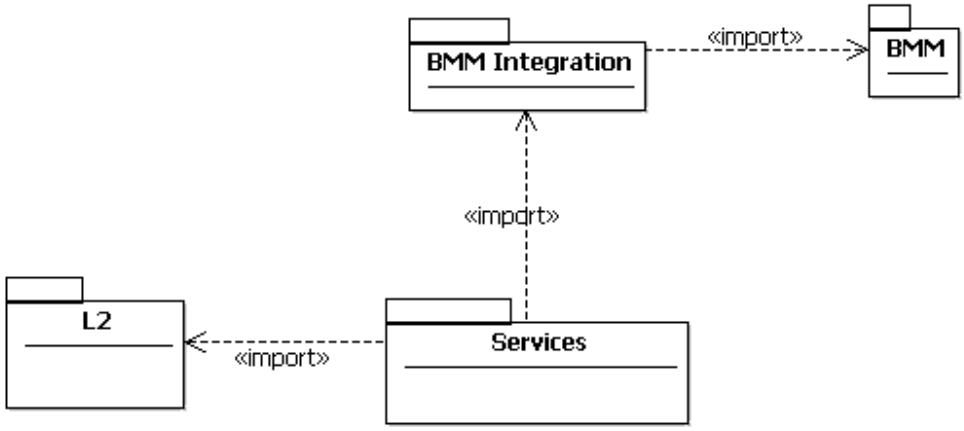

Figure 2: SoaML Compliance level SoaML:L2

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.

- UML Profile and Metamodel for Services RFP OMG document number soa/06-09-09
- UML v2.1.2 Superstructure Specification OMG document number formal/07-11-02
- UML v2.1.2 Infrastructure Specification OMG Document number formal/07-11-04
- MOF v2.0 Specification OMG document number formal/06-01-01
- MOF 2.0/XMI Mapping v2.1.1 Specification OMG document number formal/07-12-01
- UML Profile for Modeling QoS and Fault Tolerence v1.1 Specification OMG document number formal/08-04-05
- Business Process Definition Metamodel v1.0 Beta 1 Specification OMG document number dtc/07-07-01
- Business Motivation Model v1.0 Specification OMG document number formal/08-08-02
- Ontology Definition Metamodel v1.0 Beta 2 Specification OMG document number ptc/07-09-09

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).

eXtended 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.

Symbols

There are no symbols defined in this specification.

SoaML UML Profile Specification

Executive Overview

Service Oriented Architecture (SOA) is a way of organizing and understanding organizations, communities and systems to maximize agility, scale and interoperability. The SOA approach is simple – people, organizations and systems provide services to each other. These services allow us to get something done without doing it ourselves or even without knowing how to do it - this provides us with efficiency and agility. 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 a capability offered through a well-defined interface and available to a community (which may be the general public). SOA is an architectural paradigm for defining how people, organizations and systems provide and use services to achieve results. SoaML as described in this specification provides a standard way to architect and model SOA solutions using the Unified Modeling Language® (UML®). The profile uses the built-in extension mechanisms of UML to define SOA concepts in terms of existing UML concepts. SoaML can be used with current "off the shelf" UML tools but some tools may offer enhanced, SOA specific, support with the compatible SoaML metamodel.

An architectural and business focused approach to SOA

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

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

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

Due to the business and architectural focus of SoaML it is important to understand that terms like *service* can have both a business and technology interpretation. We can understand how a service of our enterprise, perhaps to sell books, is related to a component of our web infrastructure – perhaps a web service to process book orders. Concepts like "Service Contract", "Participant" and "Service" which you will see in this specification are intended for, and work well for, both business and systems architectures using the SOA paradigm.

Top down and bottom-up SOA

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

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

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

- Designing services architectures that specify a community of interacting participants, and the service contracts that reflect the agreements for how they intend to interact in order to achieve some common purpose
- Organizing individual functions into service capabilities arranged in a hierarchy showing anticipated usage dependencies.
- Using a business process to identify functional capabilities needed to accomplish some purpose as well as the roles played by participants. Processes and services are different views of the same system – one focusing on how and why parties interact to provide each other with products and services and the other focusing on what activities parties perform to provide and use those services.

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

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

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

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

Key Concepts of Basic Services

A key concept is, of course, the Service. A Service is a capability offered by one entity or entities to others using well defined "terms and conditions" and interfaces. Those entities may be people, organizations, technology components or systems – we call these Participants. Participants offer capabilities through services on Ports with the «service» stereotype. The service port is the interaction point where consumers of the service go to use that service.

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 3 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, populate 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.

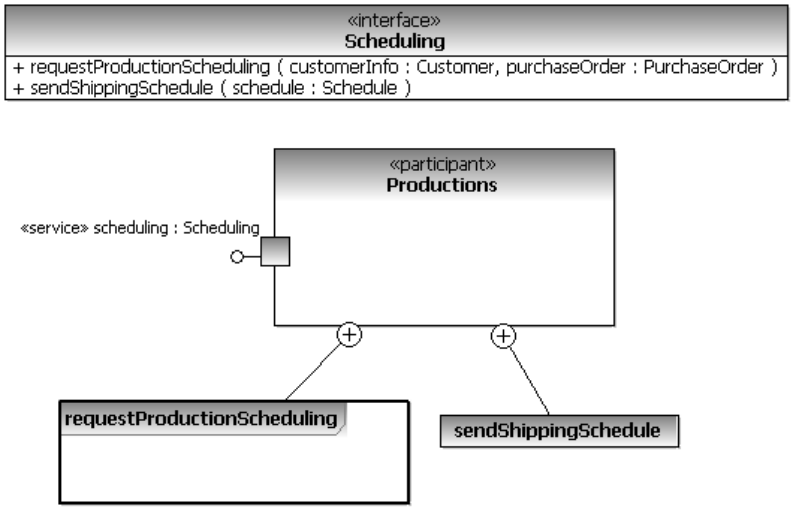

Figure 3: Services and Service Participants

Service Interfaces

Like a UML interface, a ServiceInterface can be the type of a service port. The service interface has the additional feature that it can specify a bi-directional service – where both the provider and consumer have responsibilities to send and receive messages and events. The service interface is defined from the perspective of the service provider using three primary sections: the provided and required Interfaces, the ServiceInterface class and the protocol Behavior.

- **The provided and required Interfaces** are standard UML interfaces that are realized or used by the ServiceInterface. The interfaces that are realized specify the provided capabilities, the messages that will be received by the provider (and correspondingly sent by the consumer). The interfaces that are used by the ServiceInterface define the required capabilities, the messages or events that will be received by the consumer (and correspondingly sent by the provider). Typically only one interface will be provided or required, but not always.
- **The enclosed parts of the ServiceInterface** represent the roles that will be played by the connected participants involved with the service. The role that is typed by the realized interface will be played by the service provider; the role that is typed by the used interface will be played by the consumer.
- **The Behavior** specifies the valid interactions between the provider and consumer – the communication protocol of the interaction, without specifying how either party implements their role. Any UML behavior specification can be used, but interaction and activity diagrams are the most common.

The capabilities and needs of a Service or Request (see below) are defined by its 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 capabilities through realized interfaces
- The needs or capabilities required of consumers in order to use the service through the used interfaces
- A detailed specification of each capability using an operation; including its name, preconditions, post conditions, inputs and outputs and any exceptions that might be raised
- Any protocol or rules for using the capabilities or how consumers are expected to respond and when through an owned behavior
- Rules for how the service must be implemented by providers
- Constraints that can determine if the service has successfully achieved its intended purpose

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 provides the information service providers need in order to implement the service.

Figure 4 shows a generic or archetype ServiceInterface. Interface1 is the provided interface defining the capabilities, while Interface2 is the required interface defining what consumers are expected to do in response to service requests. The ServiceInterface has two parts, part1 and part2 which represent the endpoints of connections between consumer requests and provider services. Part1 has type Interface1 indicating it represents the provider or service side of the connection. Part2 has type Interface2 indicating it represents the consumer or Request side of the connection.

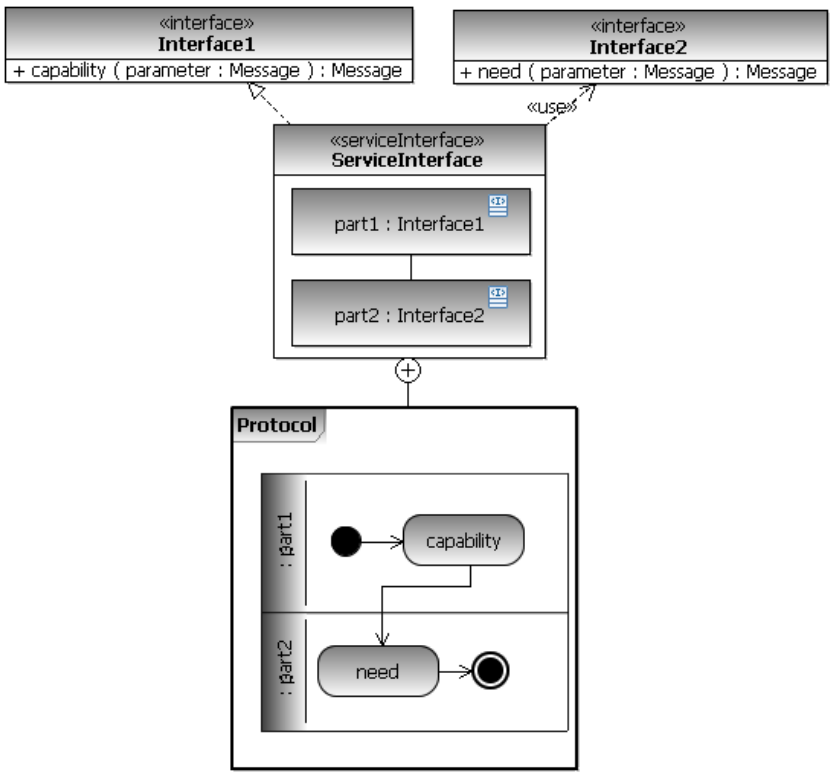

Figure 4: A generic service interface

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

Participants and Service Ports

Participants represent software components, organizations, systems or individuals that provide and use services. Participants define types of components by the roles they play in services architectures and the services they provide and use. For example, the figure to the right, illustrates a Manufacturer participant that offers a purchasing service. Participants provide capabilities through Service 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 interaction point where a classifier interacts with other classifiers (see Figure 5). A port typed by a ServiceInterface, is known as a s*ervice port*.

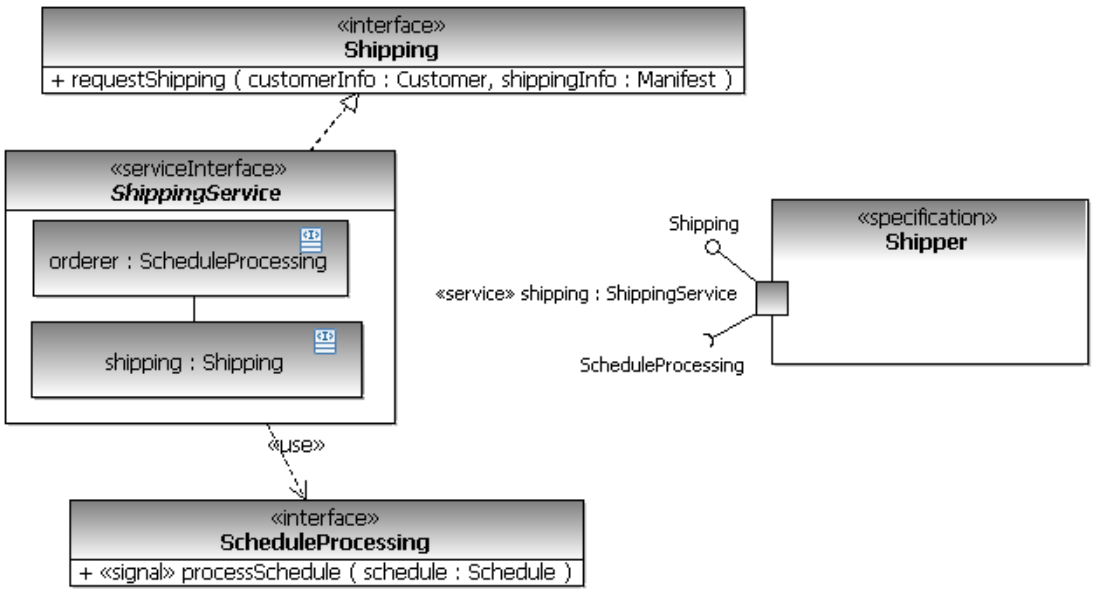

Figure 5: Example Participant with a Service Port

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

The Service Request

Just as we want to define the services provided by a participant using a service 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» as shown in Figure 6.

Figure 6: A Participant with Services and Requests

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

The OrderProcessor participant example, above, shows that it provides the "purchasing" service using the "Purchasing" ServiceInterface and Requests a "shipping" service using

the “ShippingService” ServiceInterface. Note that this request is the conjugate of the service that is offered by a Shipper, as shown in Figure 5.

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

Key Concepts of the Services Architecture

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

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

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

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

A *community* ServicesArchitecture (see Figure 7) is defined using a UML Collaboration.

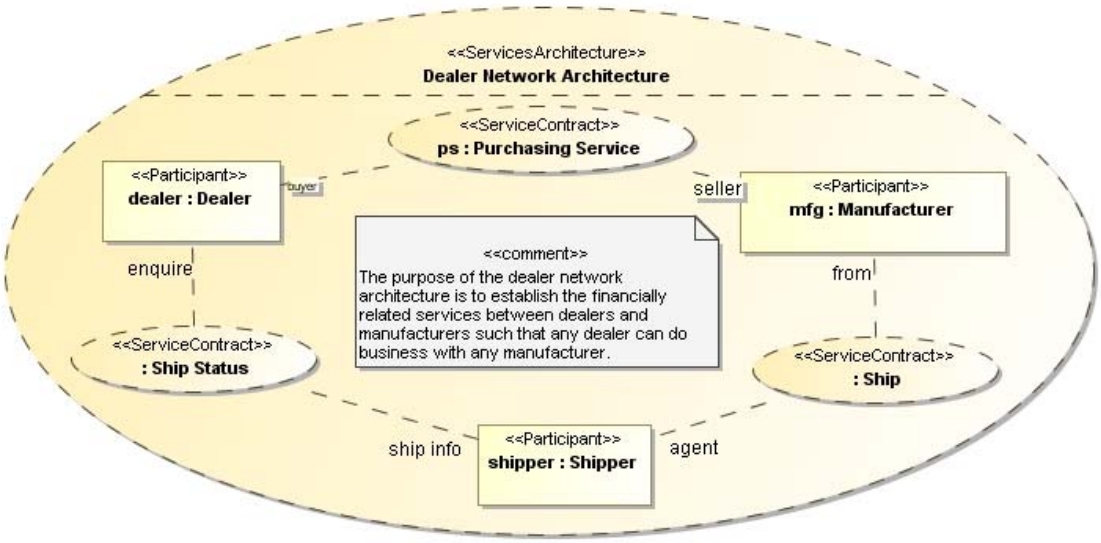

Figure 7: Example community services architecture with participant roles and services

The purpose of 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 *participant* ServicesArchitecture specifies the architecture for a particular Participant. Within a participant, where there is a concept of "management" exists, a services architecture illustrates how sub-participants and external collaborators work together and would often be accompanied by a business process. A ServicesArchitecture (both community and participant) may be composed from other services architectures and service contracts. As shown in Figure 8, 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.

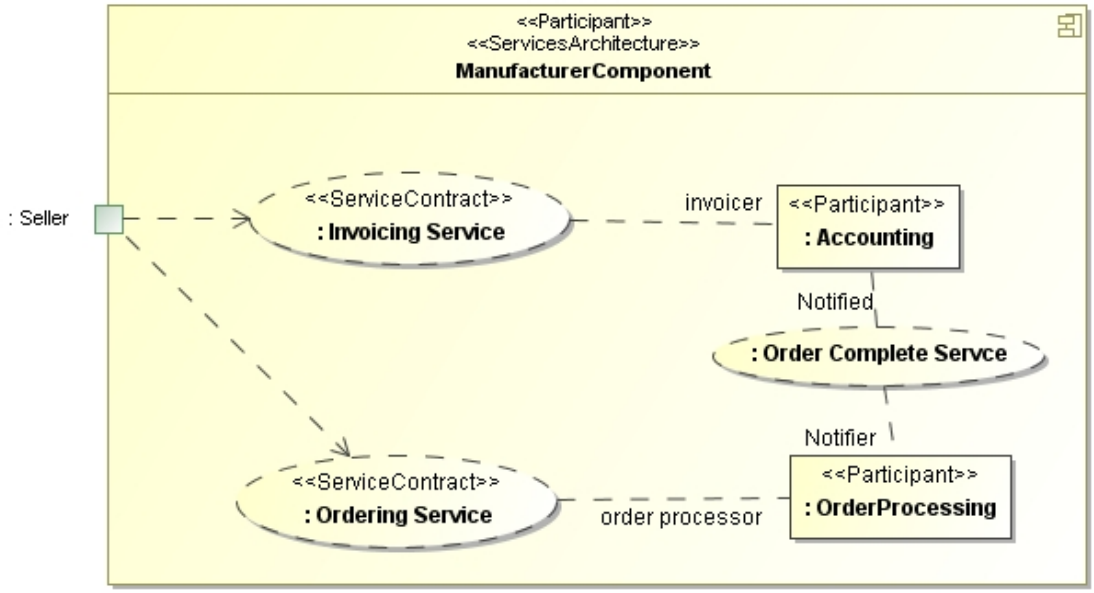

Figure 8: Example Services architecture for a participant

Figure 8 illustrates the services architecture for a "Manufacturer" participant. It indicates that this architecture consists of a number of other participants interacting through service contracts. The Manufacture services architecture would include a business process that specifies how these participants interact in order to provide a purchasing service.

Service Contracts

A key part of a service is the ServiceContract (see Figure 9).

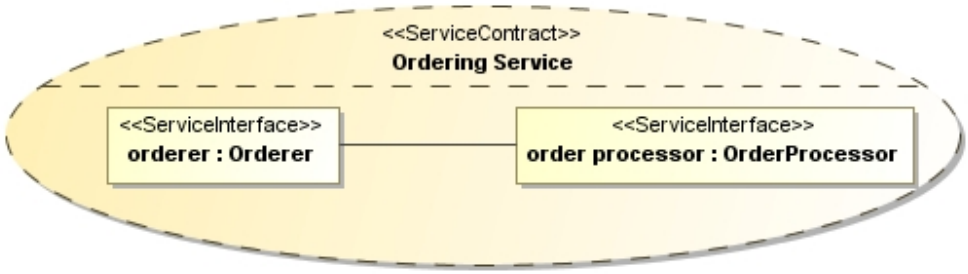

Figure 9: Example ServiceContract

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 which includes all the information, choreography and any other "terms and conditions" of the service. A ServiceContract is binding on *both* the providers and consumers of that service. The basis of the service contract is also a UML collaboration that is focused on the interactions involved in providing a service. A participant plays a role in the larger scope of a ServicesArchitecture and also plays a role as the provider or user of services specified by ServiceContracts.

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

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

A ServiceContract choreography is a UML Behavior such as may be shown on an interaction diagram or activity diagram that is owned by the ServiceContract (Figure 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.

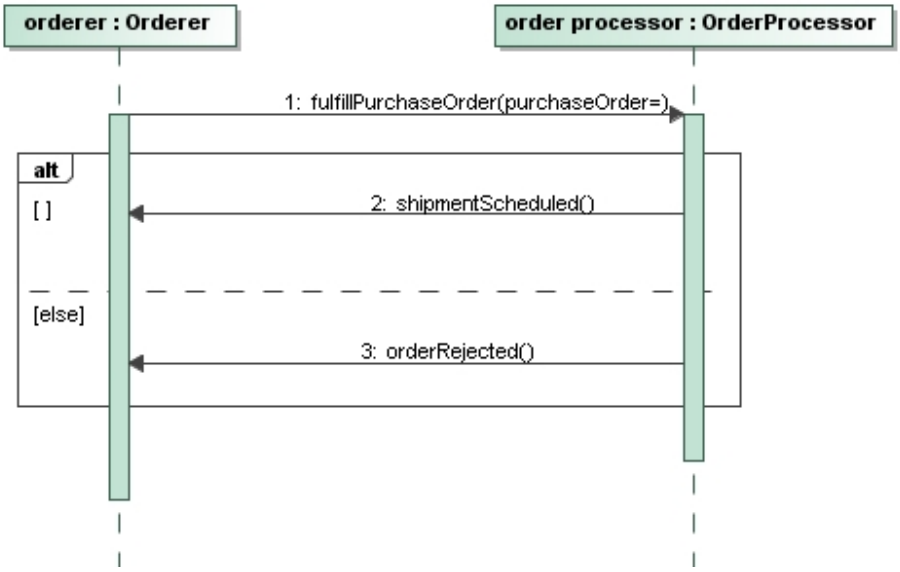

Figure 10: Example choreography

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

It should also be noted here that it is the expectation of SoaML that services may have communications going both ways – from provider to consumer and consumer to provider and that these communications may be long-lived and asynchronous. The simpler concept of a request-response function call or invocation of an 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 and participants.

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

One of the important capabilities of SOA is that it can work "in the large" where independent entities are interacting across the Internet to internal departments and processes. This suggests that there is a way to decompose a ServicesArchitecture and visualize how services can be implemented by using still other services. A participant can be further described by its internal architecture, the services architecture of the participant. Such an architecture can also use internal or external services, participants, business processes and other forms of implementation. Our concern here is to show how the internal structure of a service participant is described using other services. This is done by defining a *participant* ServicesArchitecture for participants in a more granular (larger scale) services architecture as is shown in Figure 7 and Figure 8.

The specification of an 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.

Service Capability

Services architectures and service contracts provide a formal way of identifying the roles played by parties, their responsibilities, and how they are intended to interact in order to meet some objective. But it is also useful to express the organization of a set of related services based on their capabilities in a "Participant agnostic" way.

ServiceCapabilities represent an abstraction of the services of some system regardless of the Participants that might provide and consume them. This element allows for the specification of services without regard for the how a particular service might be implemented and subsequently offered to consumers by a Participant. It allows architects to analyze how services are related and how they might be combined to form some larger capability prior to allocation to a particular Participant.

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

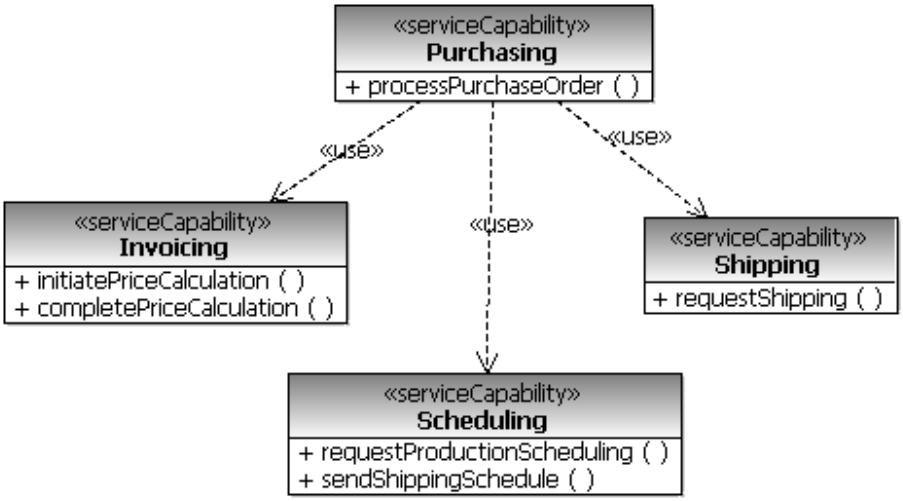

Figure 11: Service Capabilities needed for processing purchase orders

Each service 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 11 depicts the ServiceCapabilities that have been identified as needed for processing purchase orders.

ServiceCapabilities represent the services or capabilities of some system or architecture. These ServiceCapabilities may then be linked to the Participants that actually provide the capabilities through UML2 Realizations as shown in Figure 12. In this figure, the Invoicing service capability is realized by a new Invoicer participant whereas the Scheduling service capability is realized by an existing Productions participant. This separation of Service Capability from Participant allows for separation of the concerns associated with how services might be related from the concerns associated with how a particular capability might be implemented. Realization provides the traceability of the capability to the Participant that realizes or provides it.

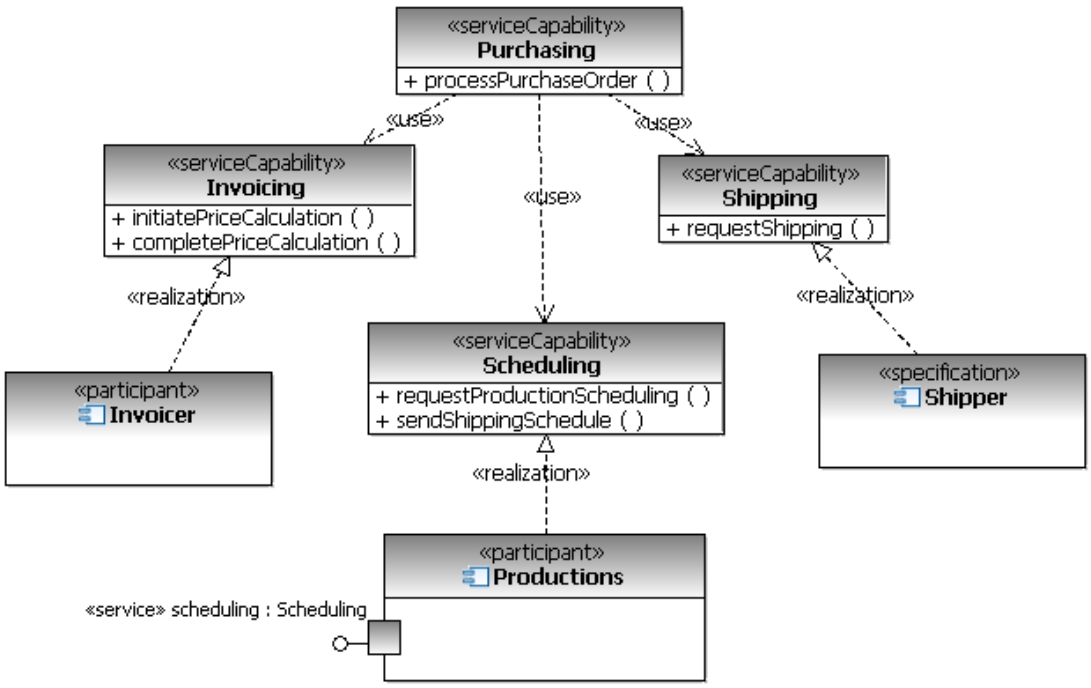

Figure 12: Realizing the order processing service capabilities

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 to 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 (for example a ServicesContract) may realize the BMM Motivation concept of *motivation realization*. 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 by an IT solution based on an 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. ServiceCapabilities, 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.

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.

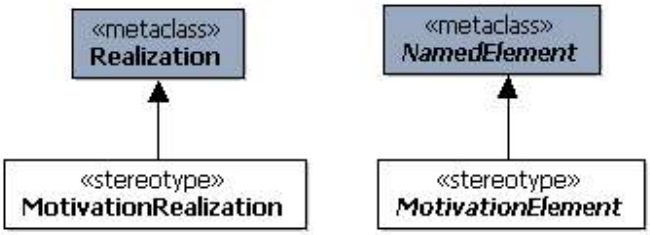

Figure 13: **BMM Integration**

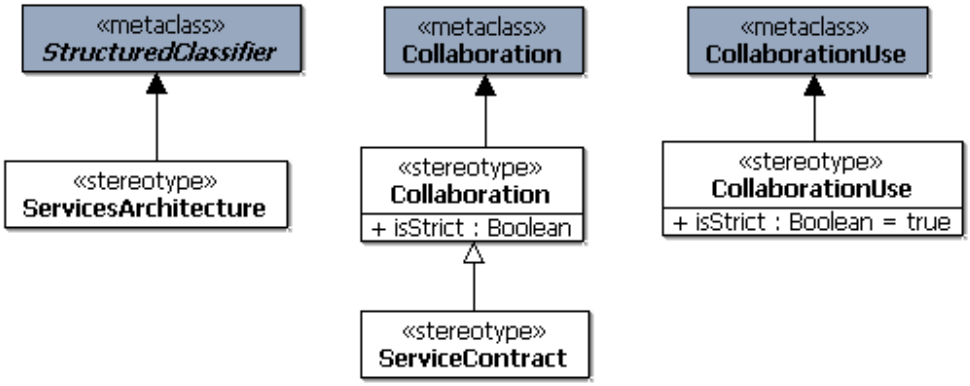

Figure 14: Contracts

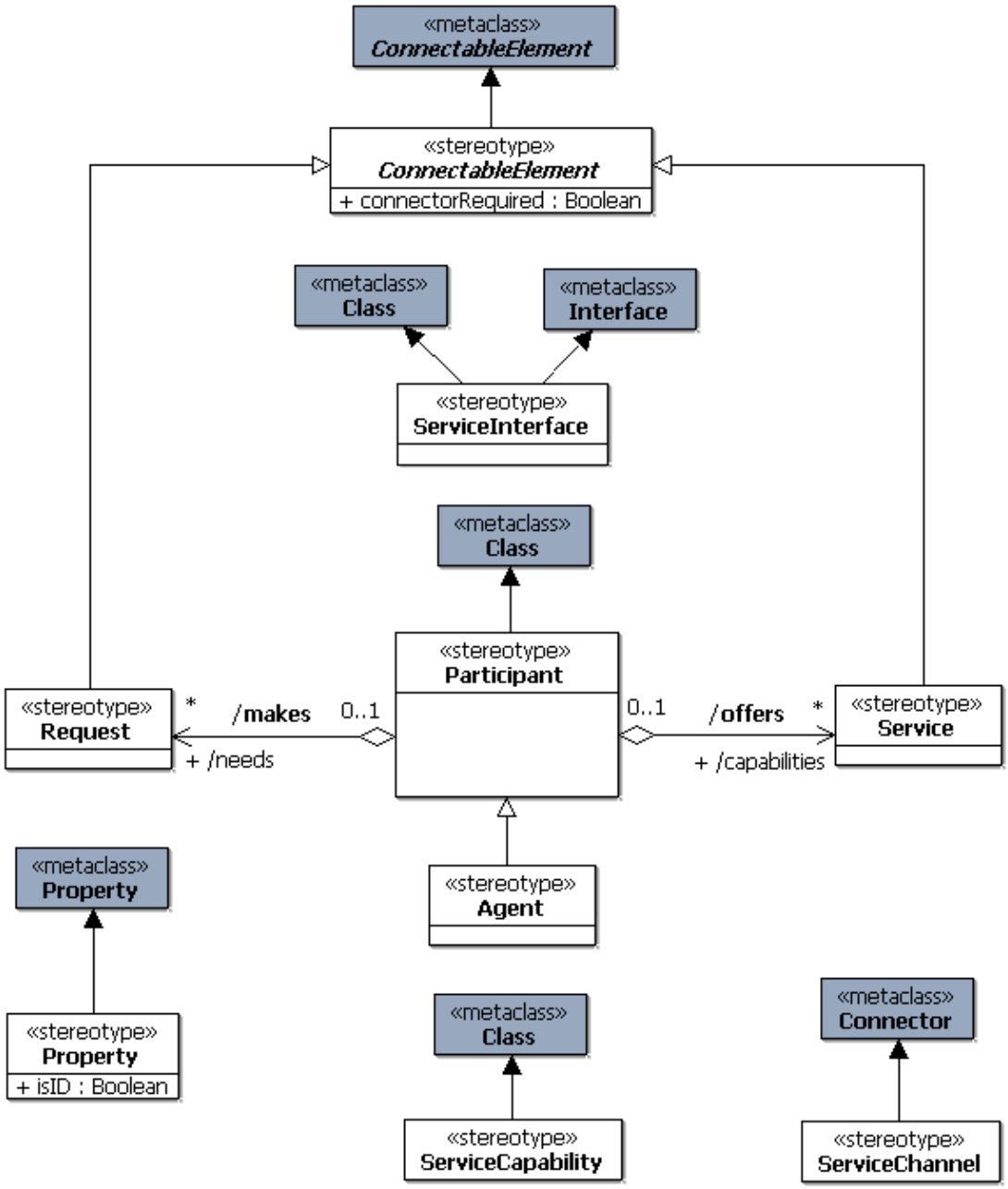

Figure 15: Services

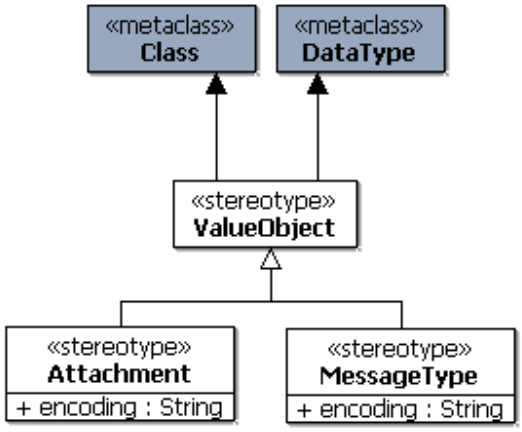

Figure 16: **Service Data**

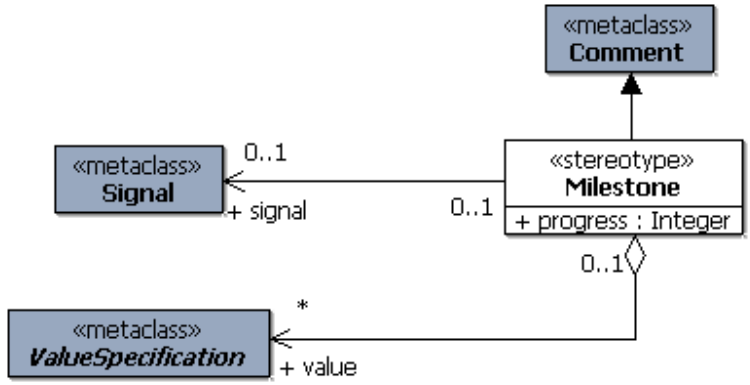

Figure 17: **Milestones**

Stereotype Descriptions

Agent

An Agent is a classification of autonomous entities that can adapt to and interact with their environment. It describes a set of agent instances that have features, constraints, and semantics in common.

Generalizes

- Participant

Description

In general, agents can be software agents, hardware agents, firmware agents, robotic agents, human agents, and so on. While software developers naturally think of IT systems

as being constructed of only software agents, a combination of agent mechanisms might in fact be used from shop-floor manufacturing to warfare systems. [2]

These properties are mainly covered by a set of core *aspects* each focusing on different viewpoints of an agent system. Even if these aspects do not directly appear in the SoaML metamodel, we can relate them to SoaML-related concepts.
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 ContractFulfillment (CollaborationUse) indicates which roles are interacting (i.e., which parts they play) in the contract. 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 Behaviours in the form of an Interaction, Activity, StateMachine, ProtocolStateMachine, or OpaqueBehavior.
- **Organization/Group aspect** – Agents can form social units called groups. A group can be formed to take advantage of the synergies of its members, resulting in an entity that enables products and processes that are not possible from any single individual.

[2] For a further history and description of agents, see:
http://eprints.ecs.soton.ac.uk/825/05/html/chap3.htm,
http://en.wikipedia.org/wiki/Software_agent,
http://www.sce.carleton.ca/netmanage/docs/AgentsOverview/ao.html.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

The property isActive must always be true.

Semantics

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

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

- Autonomous - is capable acting without direct external intervention. Agents have some degree of control over their internal state and can act based on their own experiences. They can also possess their own set of internal responsibilities and processing that enable them to act without any external choreography. As such, they can act in reactive and proactive ways. When an agent acts on behalf of (or as a proxy for) some person or thing, its autonomy is expected to embody the goals and policies of the entity that it represents. In UML terms, agents can have classifier behavior that governs the lifecycle of the agent.

- Interactive - communicates with the environment and other agents. Agents are interactive entities because they are capable of exchanging rich forms of messages with other entities in their environment. These messages can support requests for services and other kinds of resources, as well as event detection and notification. They can be synchronous or asynchronous in nature. The interaction can also be conversational in nature, such as negotiating contracts, marketplace-style bidding, or simply making a query. In the Woodridge-Jennings definition of agency, this property is referred to as *social ability*.

- Adaptive - capable of responding to other agents and/or its environment. Agents can react to messages and events and then respond in a timely and appropriate manner. Agents can be designed to make difficult decisions and even modify their behavior based on their experiences. They can learn and evolve. In the 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 an SOA system.

Participants are Components whose capabilities and needs are static. In contrast, Agents are Participants whose needs and capabilities may change over time.

In SoaML, Agent is a Participant (a subclass of Component). A Participant represents some concrete Component that provides and/or consumes services and is considered an active class (isActive=true). However, SoaML restricts the Participant's classifier behavior to that of a constructor, not something that is intended to be long-running, or represent an "active" lifecycle. This is typical of most Web Services implementations as reflected in WS-* and SCA.

Agents possess the capability to have services and Requests and can have internal structure and ports. They collaborate and interact with their environment. An Agent's classifierBehavior, if any, is treated as its 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. Is can also represented by stick "agent" man icon with the name of the agent in the vicinity (usually above or below) the icon. (Figure 18) Note: the stick icon is similar to the notation for UML actors, except that it has a wizard hat shaped as an elongated "A".

Figure 18: Agent notation.

Additions to UML 2.X

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

Attachment

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

Generalizations

- ValueObject

Description

An Attachment denotes some component of a messages which is an attachment to it (as opposed to a direct part of the message itself). In general this is not likely to be used greatly in higher level design activities, but for many processes attached data is important

to differentiate from embedded message data. For example, a catalog service may return general product details as a part of the structured message but images as attachments to the message; this also allows us to denote that the encoding of the images is binary (as opposed to the textual encoding of the main message). Attachments may be used to indicate part of service data that can be separately accessed, reducing the data sent between consumers and providers unless it is needed.

Attributes

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

Associations

No additional associations

Constraints

No additional constraints

Semantics

In an SOA supporting some business, documents may represent legally binding artifacts defining obligations between an enterprise and its partners and clients. These documents must be defined in a first class way such that they are separable from the base message and have their own identity. The 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 is a specialization of ValueObject used 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 19 shows an InvoiceContent Attachment to the Invoice MessageType. This attachment contains the detailed information about the Invoice.

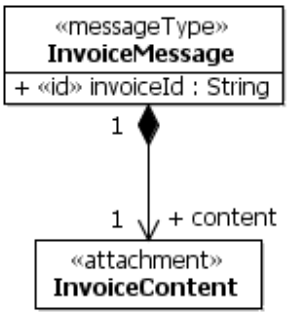

Figure 19: The InvoiceContent Attachment

Additions to UML 2.x

Extends UML 2.x to distinguish message attachments from other message properperties.

CollaborationUse

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

Extends Metaclass

- CollaborationUse

Description

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

Attributes

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

Associations

No new associations.

Constraints

No new constraints.

Semantics

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

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

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, or
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

Behavioral compatibility in CollaobrationUses is a semantic variation point. In general, the actions should be invoked in the same order, but how this is determined based on flow analysis is not specified.

Notation

No new notation.

Examples

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

Figure 20 shows a ShipingService 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.

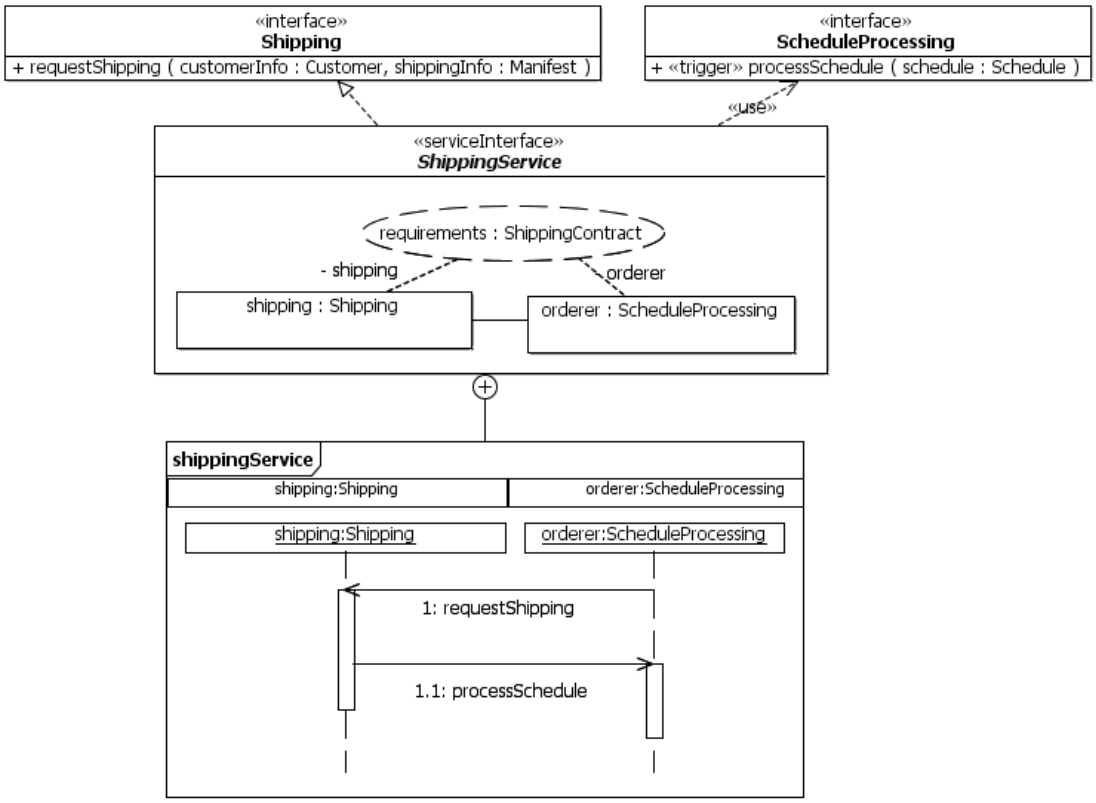

Figure 20: Fulfilling the ShippingContract ServiceContract

Figure 21 shows a Participant that assembles and connects a number of other Participants in order to fulfill the Process Purchase Order ServicesArchitecture. In this case, the roles in the ServicesArchitecture are typed by either ServiceInterfaces or Participants and the architecture specifies the expected interaction between those Participants in order to accomplish some desired result.

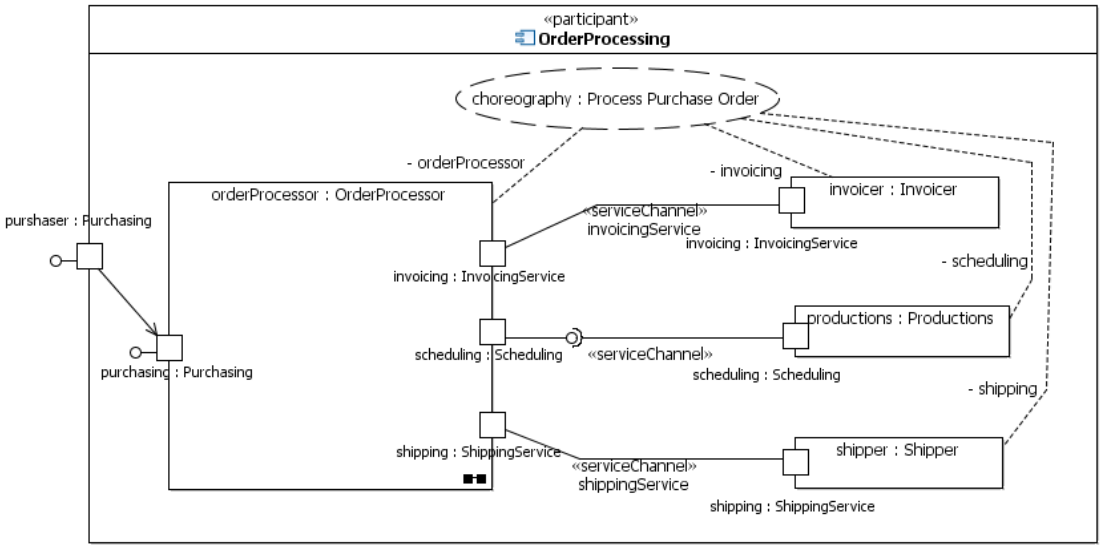

Figure 21: Fulfilling the Process Purchase Order Contract

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

Additions to UML 2.X

CollaborationUse extends UML 2.x CollaborationUse to include the isStrict property.

ConnectableElement

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

Extends Metaclass

- ConnectableElement

Description

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

Attributes

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

Associations

No additional Associations.

Constraints

No additional constraints.

Semantics

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

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

Notation

No additional Notation.

Additions to UML 2.x

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

MessageType

The specification of information exchanged between service consumers and providers.

Specializes

- ValueObject

Description

A MessageType is a kind of ValueObject 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 ValueObjects 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.

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
[4] All ownedAttributes of a MessageType must be PrimitiveType, DataType, ValueObject, or another MessageType or a reference to one of these types.
[5] If an Operation has an ownedParameter whose type is a MessageType, then the Operation can have at most one in, one out, and one exception parameter, and all ownedParameters must be PrimitiveType or MessageType. Such an operation cannot have an inout or return parameter.

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 requried 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.

Notation

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

Examples

Figure 22 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.

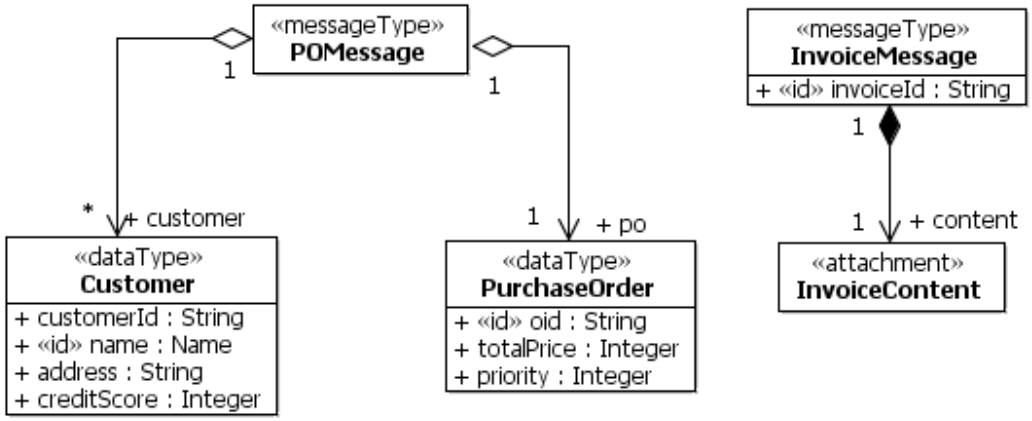

Figure 22: MessageTypes in Purchase Order Processing

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 23 shows two examples of the Purchasing Interface it 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 traditional Remote Procedure Call (RPC) style which supports multiple inputs, outputs and a return value. The choice to use depends on modeler preference and possibly the target platform. Some platforms such as Web Services and WSDL require message style

parameters. 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 which could cause interoperability problems in the runtime platform.

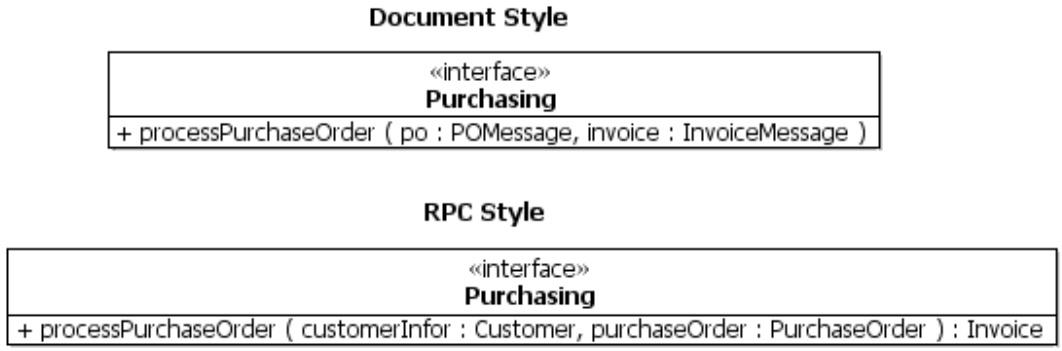

Figure 23: Document and RPC Style service operation parameters

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 UML 2.X

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

Milestone

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

Extends Metaclass

- Comment

Description

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

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

Agent negotiations, such a progress measurement could be a useful criterion for the choice.

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

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

Attributes

- progress: Integer The progress measurement.

Associations

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

Constraints

No new constraints.

Semantics

A Milestone can be understood as a “mythical” Signal. A mythical Signal is a conceptual signal that is sent from the behavior every time a point connected to the Milestone is passed during execution. The signal is sent to a conceptual observer outside the system that is able to record the origin of the signal, the signal itself and its progress value.

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

Notation

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

Examples

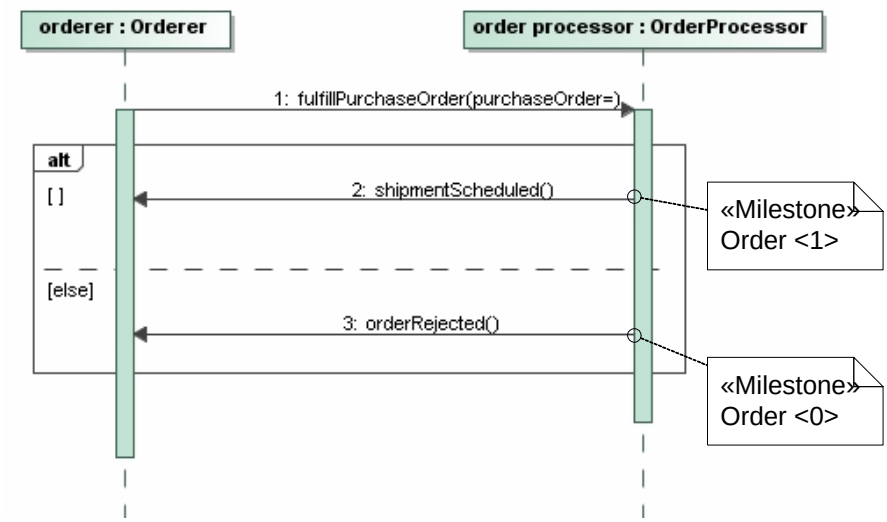

Figure 24: Milestones on Ordering Behavior

In Figure 24 we have taken the ordering behavior from Figure 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 UML 2.x

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

Participant

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.

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.

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

The full scope of an 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 and Request ports.
[2] All the capabilities of a Participant must be provided through some Service.
[3] All the needs of a Participant must be consumed through some Request.
[4] Note that the technology implementation of a component implementing a participant is not bound by the above rules, the connections to a participant components "container" and other implementation components may or may not use services.

Semantics

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

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

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. Constrast 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.

Semantic Variation Points

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

Notation

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

Examples

Figure 25 shows an OrderProcessor Participant which provides the purchasing Service. This service provides the Purchasing Interface which has a single capability modeled as the processPurchaseOrder Operation. The OrderProcessor Participant also has Requests for invoicing, scheduling and shipping. Participant OrderProcessor provides a method activity, processPurchaseOrder, for its processPurchaseOrder service operation. This activity defines the implementation of the service capability.

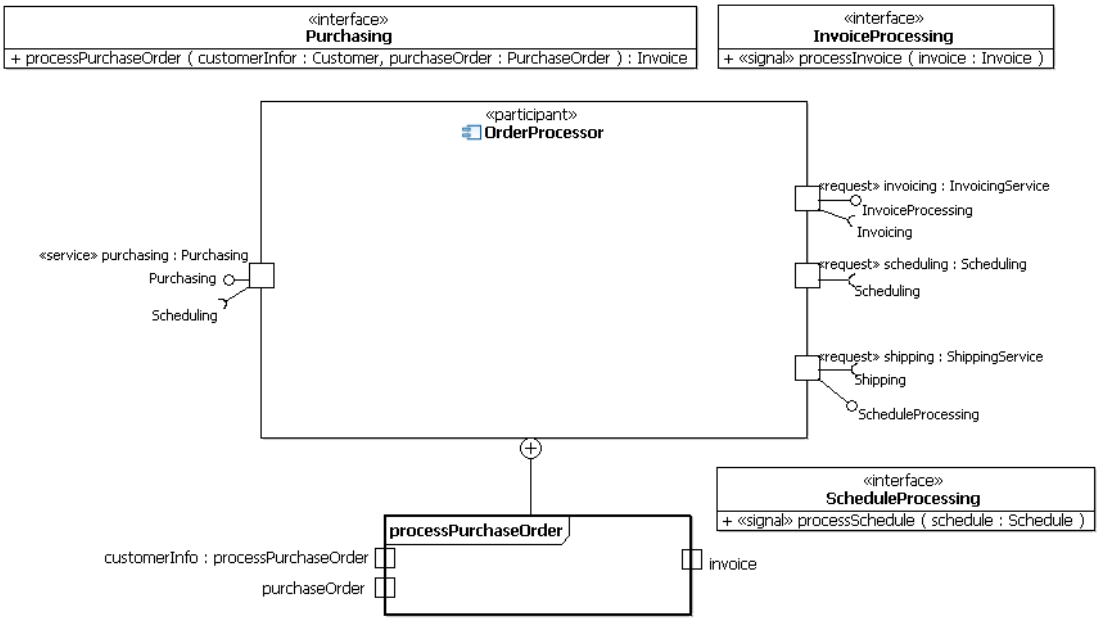

Figure 25: The OrderProcessor Participant

Figure 26 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.

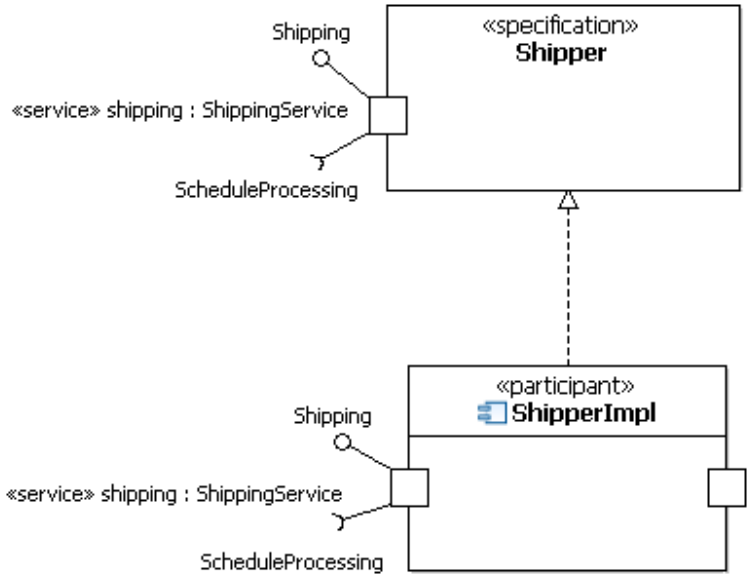

Figure 26: The Shipper specification and the ShipperImpl realization Participants

Figure 27 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.

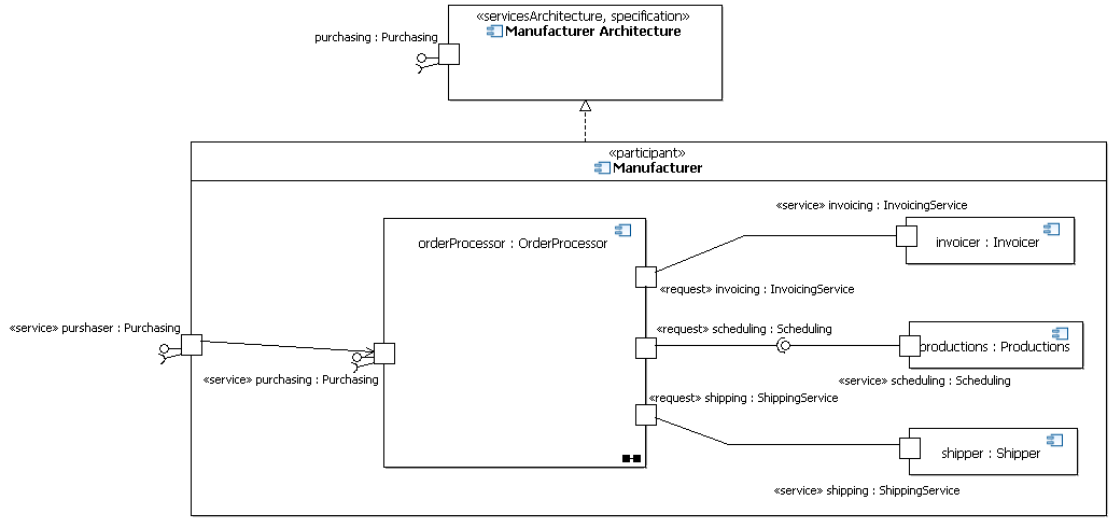

Figure 27: The Manufacturer Participant

Additions to UML 2.x

Participant is a stereotype of UML2 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.

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”.

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 one capable of identifying instance of the classifier. Compound identifiers can be created by using a Class or DataType to define a property with isID=true.

Notation

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

Examples

Figure 22 show an example of both data and message types with identifying properties.

Figure 28 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.

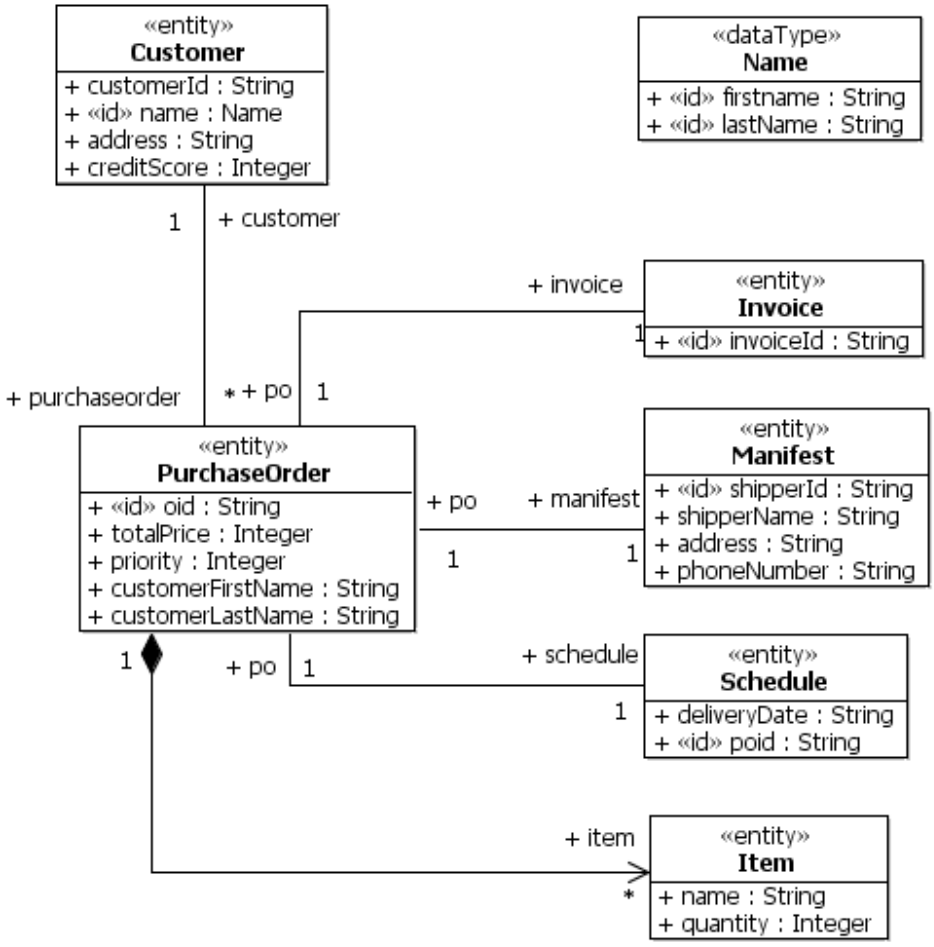

Figure 28: **Example entities from the CRM domain model**

Additions to UML 2.x

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

Request

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

Extends Metaclass

- ConnectableElement

Description

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

A 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 ConnectableElement and changes how provided and required interfaces for the ConnectableElement are interpreted. The capabilities consumed through the Request, its required interfaces, are derived from the interfaces realized by the service's ServiceInterface. The capabilities provided by consumers in order to use the service, its provided interfaces, are derived from the interfaces used by the service's ServiceInterface. These are the opposite of the provided and required interfaces of a Port or Service and indicate the use of a Service rather than a provision of a service. Since the provided and required interfaces are reversed, a request is the *use of* the service interface – or 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. Requisition and Service effectively give Ports a direction indicating whether the capabilities defined by a ServiceInterface are used or provided.

Attributes

No new attributes.

Associations

No new associations

Constraints

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

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 it's 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 there is no provided Interface and no protocol.

Notation

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

Examples

Figure 29 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.

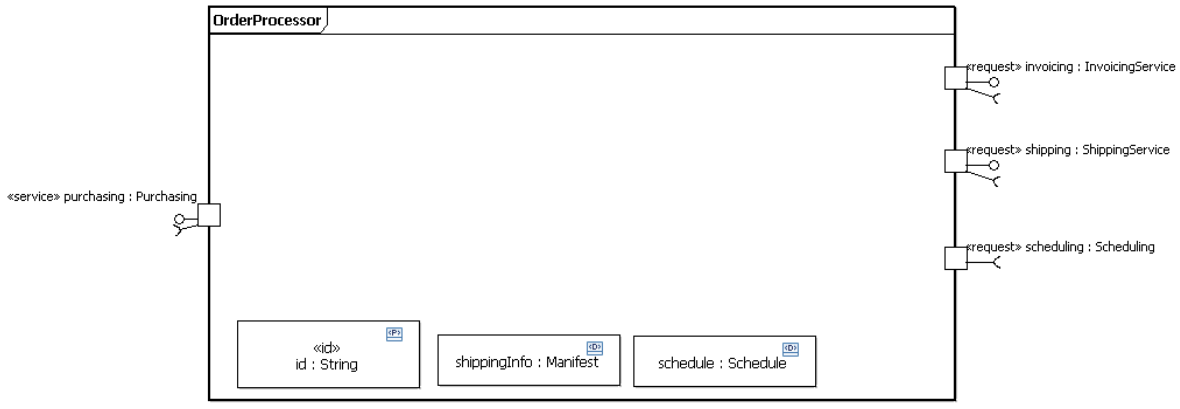

Figure 29: Requests of the OrderProcessor Participant

The invoicing Request is typed by the InvoicingService ServiceInterface. The scheduling Request is typed by the Scheduling Interface. This is an example of a simple Request that

specifies simply a list of needs. It is very similar to a Reference in SCA. See section ServiceInterface for details on these service interfaces. See section Service for examples of Participants that provides services defined by these service interfaces.

Additions to UML 2.x

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

Service

A Service is a capability offered by one entity or entities to others using well defined terms, conditions and interfaces. The service stereotype of a port defines the connection point through which a Participant offers a service to clients.

Extends Metaclass

- ConnectableElement

Description

A Service is a mechanism by which a provider Participant makes available capabilites that meet the needs of consumer Requests as defined by ServiceInterfaces, Interfaces and ServiceContracts. A Service is usually represented by a UML Port on a Participant stereotyped as a «service», though which the service is delivered. This will be refered to as a service port. A Service may also designate a role in a ServiceContract.

A Service port may include the specification of the value offered to another, and the offer to provide value to another. A service port is the visible point at which consumer Requests are connected to providers, and through which they interact in order to produce some real world effect. A service defines a capability that represents some functionality created to address a potential need, and the point of access to bring that capability to bear in an execution context. The capability is defined by a Service owned by a Participant and typed by a ServiceInterface.

A Service may also be viewed as some capability or set of related capabilities provided by a provider Participant and consumed by some consumer Participants that 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. The capabilities required of consumers in order to use the service, its required interfaces, are derived from the interfaces used by the service's ServiceInterface. These are the same as the provided and required interfaces of the Port that is a generalization of Service.

Attributes

No new attributes.

Associations

No New associations.

Constraints

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

Semantics

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

If the type of a 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 port is defined as a role within a ServiceContract – the service port (and participant) is bound by the semantics and constraints of that service contract.

Notation

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

Examples

Figure 30 shows an invoicing service provided by an Invoicer Participant. In this example, the Invoicer Participant realizes the Invoicing UseCase that describes the high-

level requirements for the service provider and its services. The invoicing Service is typed by the InvoicingService ServiceInterface which defines the interface to the service. See the section ServiceInterface for further details on this ServiceInterface.

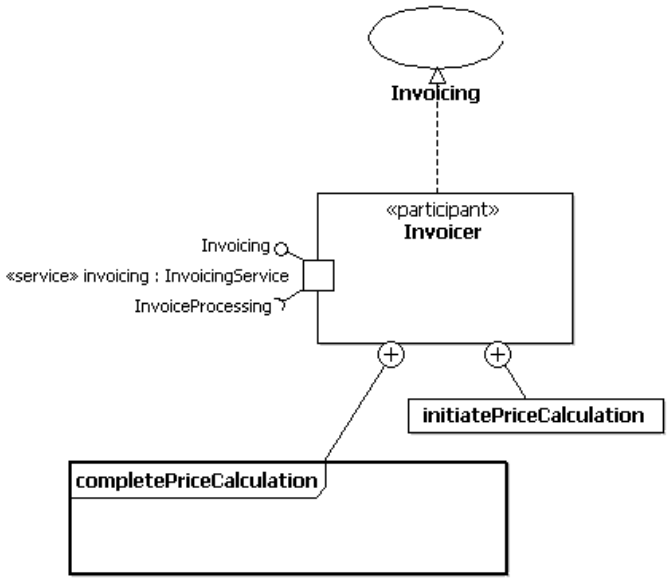

Figure 30: The invoicing Service of the Invoicer Participant

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

Figure 31 shows 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.

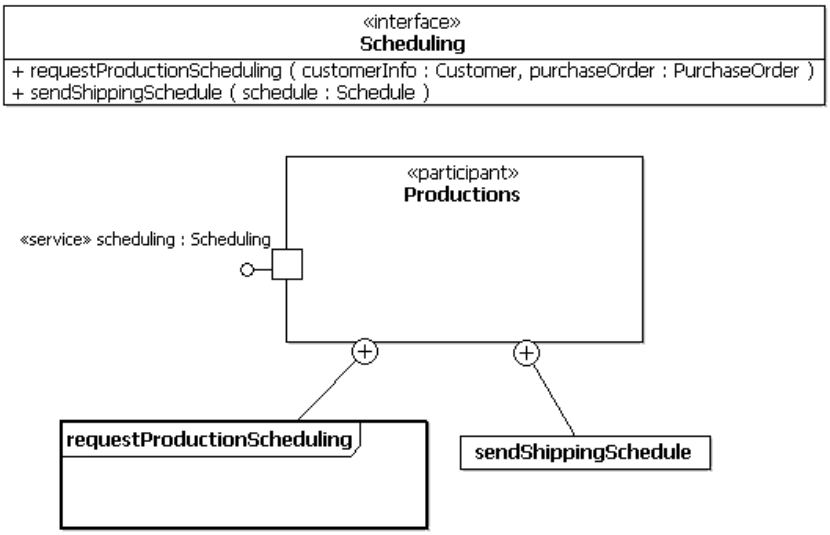

Figure 31: The scheduling Service of the Productions Participant

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

Additions to UML 2.X

ServiceCapability

A ServiceCapability specifies a capability to provide a service.

Extends Metaclass

- Class

Description

A ServiceCapability models the capability for providing 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 ServiceCapability may specify dependencies or internal process to detail how that service capability is provided including dependencies on other ServiceCapabilities. ServiceCapabilities are shown in context using a service dependencies diagram.

Attributes

No additional attributes.

Associations

No additional Associations.

Constraints

No additional constraints.

Semantics

ServiceCapabilities represent an abstraction of the services of some system or services architecture regardless of the participants that might provide and consume them. This element allows for the specification of services without regard for the how a particular service might be implemented and subsequently offered to consumers by a Participant. It allows architects to analyze how services are related and how they might be combined to form some larger capability prior to allocation to a particular Participant.

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

Each service 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 11 depicts the ServiceCapabilities that have been identified as needed for processing purchase orders.

Notation

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

Examples

For examples of ServiceCapability, see Figure 11 and Figure 12.

Additions to UML 2.x

ServiceCapability is a new stereotype used to describe service capabilities.

ServiceChannel

A communication path between Requests and services.

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
[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 Request and Service

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 capabilities. 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 satisified, Requests must be connected to compatibile Services. This does not mean the Request and Service 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 satisify 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 which 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 though 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.

Semantic Variation Points

Behavioral compatibility between Requests and Services is a semantic variation point. In general, the actions should be invoked in the same order, but how this is determined based on flow analysis is not specified.

Notation

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

Examples

Figure 32 illustrates an OrderProcessing service Participant that assembles a number of other Participants necessary to actually implement a service as a deployable runtime solution. OrderProcessing provides a purchaser service which it delegates to the purchasing service of its orderProcessor part. ServiceChannels connect the Requests to the Services the OrderProcessor needs in order to execute.

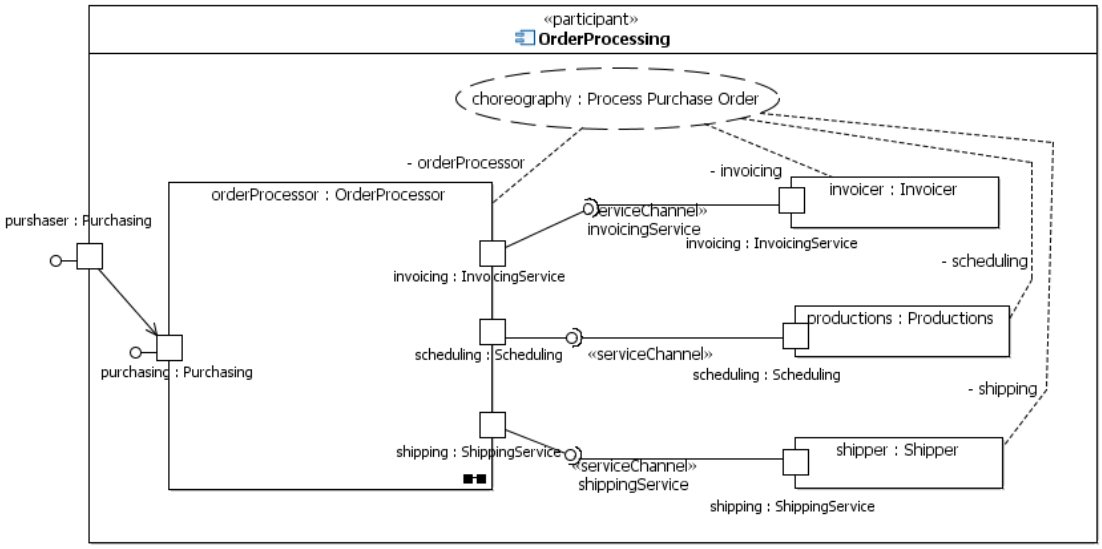

Figure 32: The OrderProcessing Participant

Additions to UML 2.X

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

ServiceContract

A ServiceContract is the formalization of a binding exchange of information, goods, or 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 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 which 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 has a type, which must be a ServiceInterface or UML Interface and usually represents a provider or consumer. The ServiceContract is a binding agreement on entities that implement the service type**.** That is, any party that "plays a role" in a Service Contract is bound by the service agreement, exchange patterns, behavior and MessageType formats. Note that there are various ways to bind to or fulfill such an agreement, but compliance with the agreement is ultimately required to participate in the service.

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

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, or
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.

Semantics

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

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

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

Examples

In the context of services modeling, ServiceContracts may be used to model the specification for a specific service. A ServicesArchitecture 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 service ServiceContract are typed by Interfaces that specify Operations that define the capabilities and needs. A ServiceInterface may fulfill zero or more ServiceContracts to indicate the requirements it fulfills.

Figure 33 is an example of a ServiceContract. The orderer and order processor participate in the contract.

Figure 33: 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.

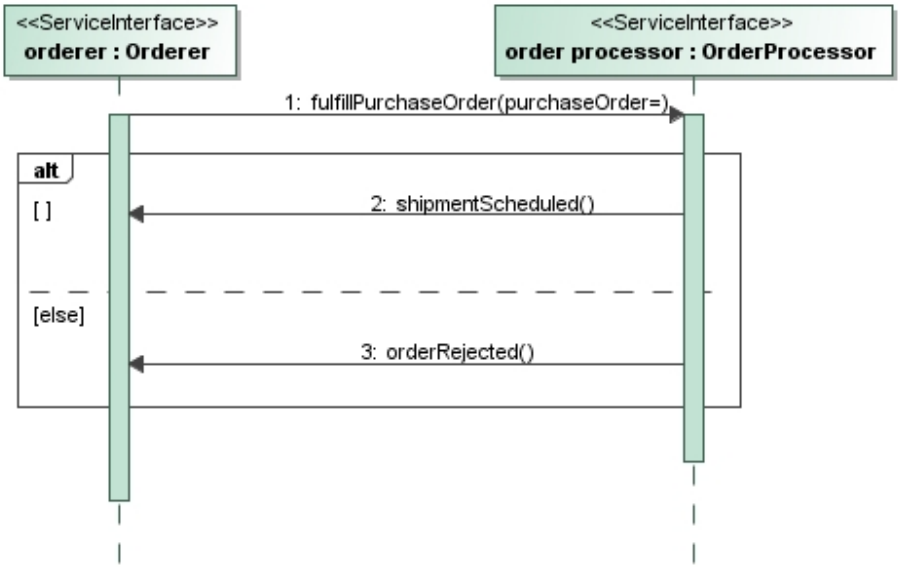

Figure 34: Ordering Service communication protocol

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

The service interfaces that correspond to the above types are:

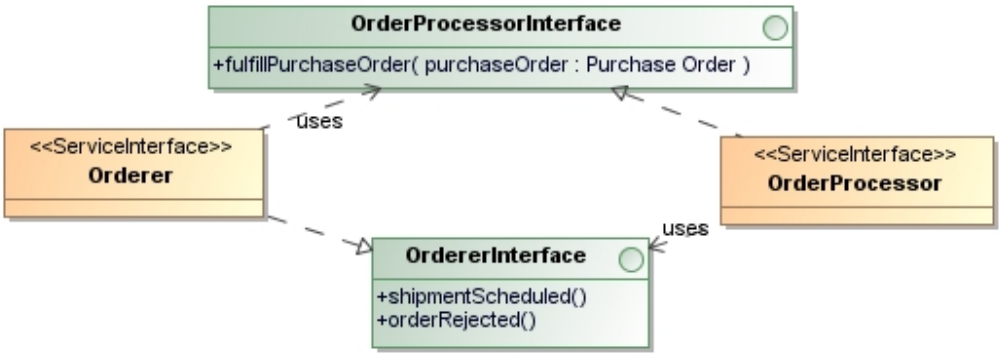

Figure 35: Service interfaces that correspond to the above

As service interface is covered in detail the explanation is not repeated here. Note that the above service interfaces are the types of the roles in the ServiceContract shown in Figure 35.

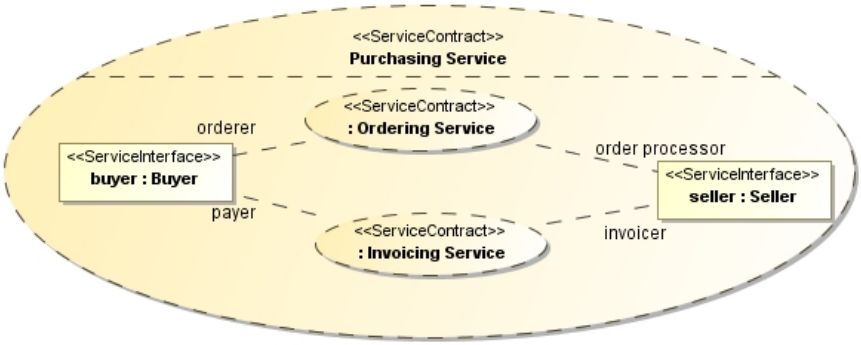

Figure 36: Compound Services

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 2 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.

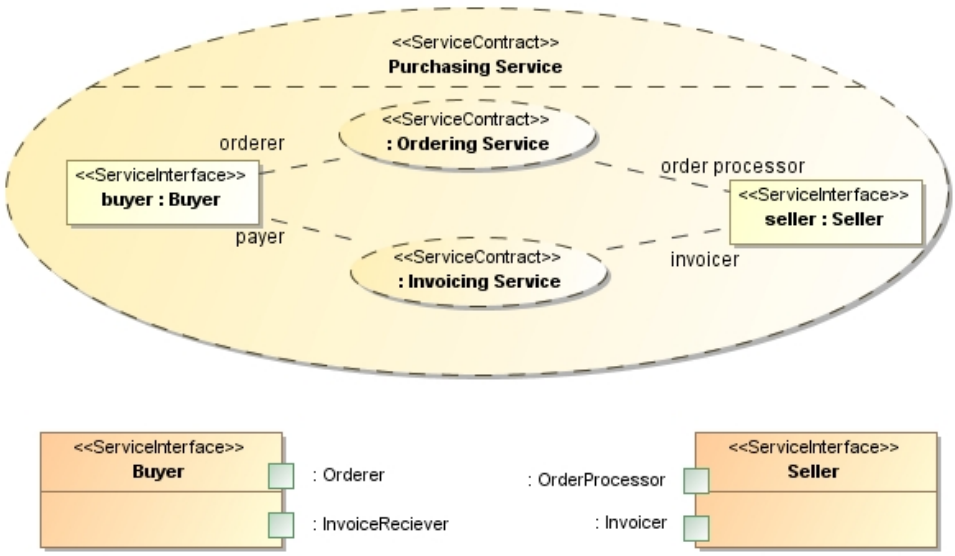

Figure 37: Service Interfaces of a compound service

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 2 ports, each corresponding to the roles played in the ordering service and invoicing service. Likewise, the buyer has 2 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 either the Collaboration notation stereotyped with «serviceContract» .

Additions to UML 2.x

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

ServiceInterface

Defines the interface to a Service or Request.

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 to interact with 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 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 responsibilitiees 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 provider services that may be used to match needs and capabilities.

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

> Note: There is somewhat of a stylistic differeance 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.
[2] A ServiceInterface must not define the methods for any its provided operations or signals.

Semantics

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

Function	Metadata
An indication of what the service does or is about	The ServiceInterface name
The capabilities 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 needs, or capabilities consumers are expected to provide 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.
The detailed specification of a capability or need including: 1. Its name, often a verb phrase indicating what it does 2. Any required or optional service data inputs and outputs 3. Any preconditions consumers are expected to meet before using the capability 4. Any post conditions consumers participants can expect, and other providers must provide upon successful use of the service 5. Any exceptions or fault conditions	Each individual capability or need of a ServiceInterface is modeled as an Operation 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.

Function	Metadata
that might be raised if the capability cannot be provided for some reason even though the preconditions have been met	
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 onwedBehavior 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.

Table 1: Information in a ServiceInterface

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

Participants specify their needs with Requests 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 Requistion may be connected to a compatible Service in an assembly of Participants through a ServiceChannel. These connected participants are the parts of the internal structure of some other Participant where they are assembled in a context for some purpose, often to implement another service, and often adhering to

some ServicesArchitecture. ServiceChannel specifies the rules for compatibility between a Request and Service. Essentially they are compatible if the needs of the Request are met by the capabilities of the Service and they are both structurally and behaviorally compatible.

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

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

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

Semantic Variation Points

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

Notation

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

Examples

Figure 38 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 or Request is similar to using a WSDL PortType or Java interface as the type of an SCA component's service or reference.

Figure 38: The StatusInterface as a simple service interface

Figure 39 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.

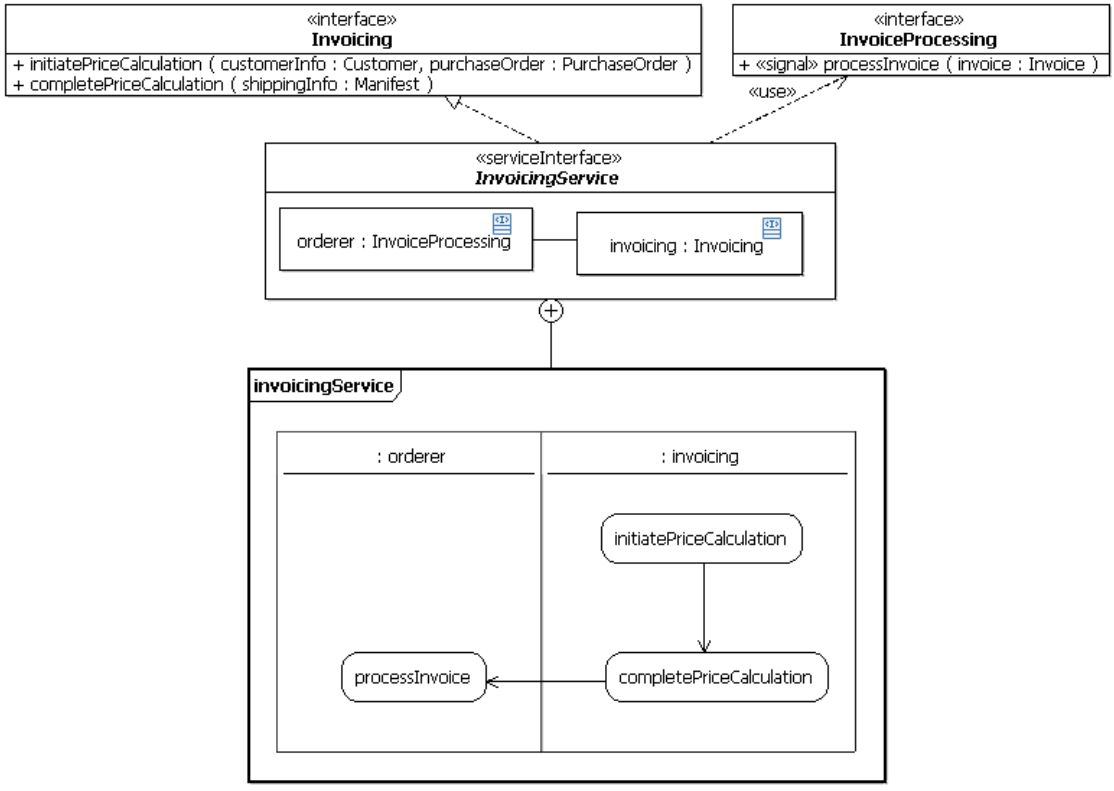

Figure 39: The InvoicingService ServiceInterface

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 40 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.

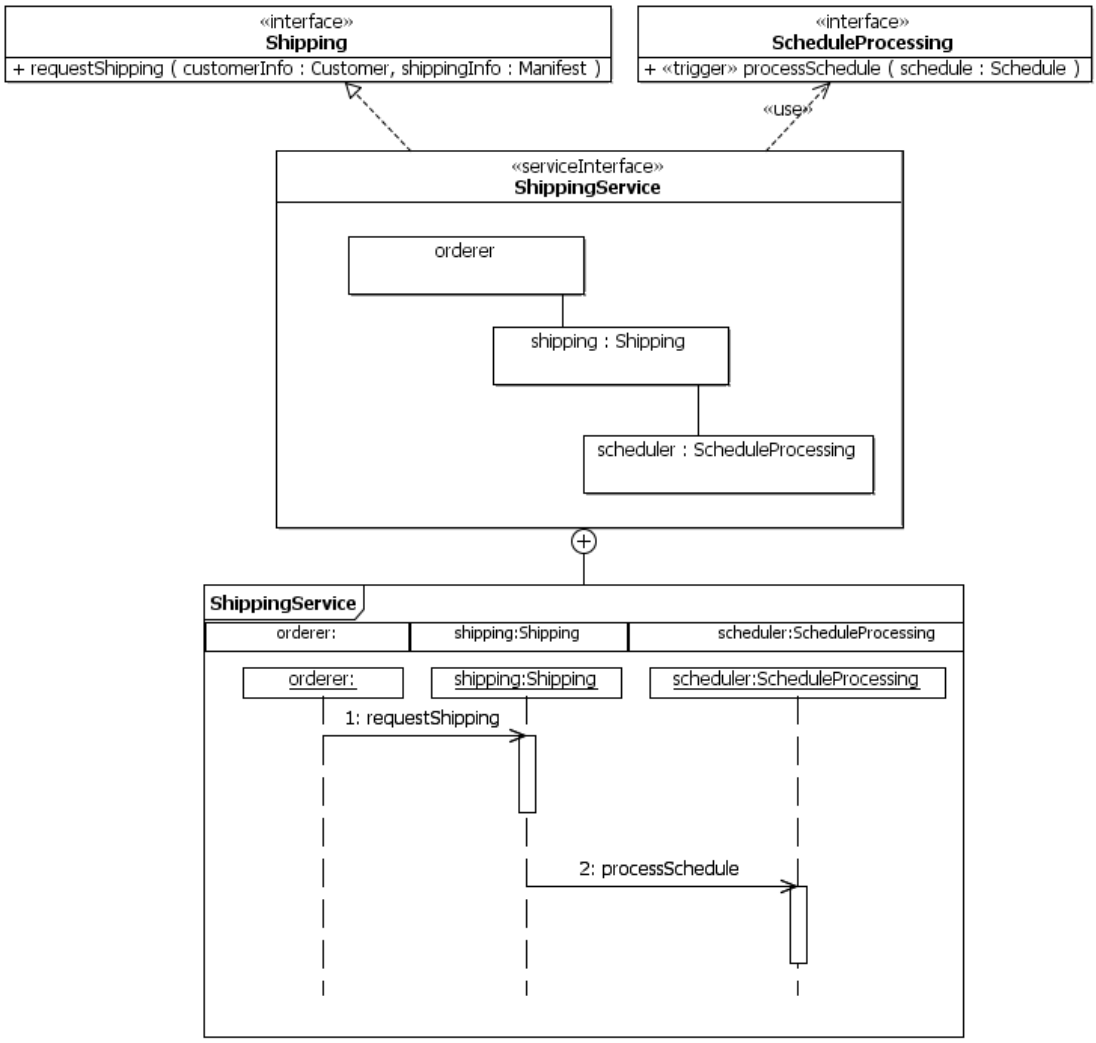

Figure 40: Another version of the ShippingService supporting additional parties

Figure 41 shows an example where aCustomer invokes the requestShipping operation of the shipping service, but aScheduler processes the schedule. This is possible because the ServiceInterface separates the request from the reply by adding the scheduler part. It is the combination of both ServiceChannels that determine compatibility with the shipping service, not just one or the other. That is, it is the combination of all the interactions through a service port that have to be compatible with the port's protocol, not each one.

Figure 42 Shows a different version of the OrderingSubsystem where aCustomer both requests the shipping and processes the order. This ServiceChannel is also valid since this version of aCustomer follows the complete protocol without depending on another part.

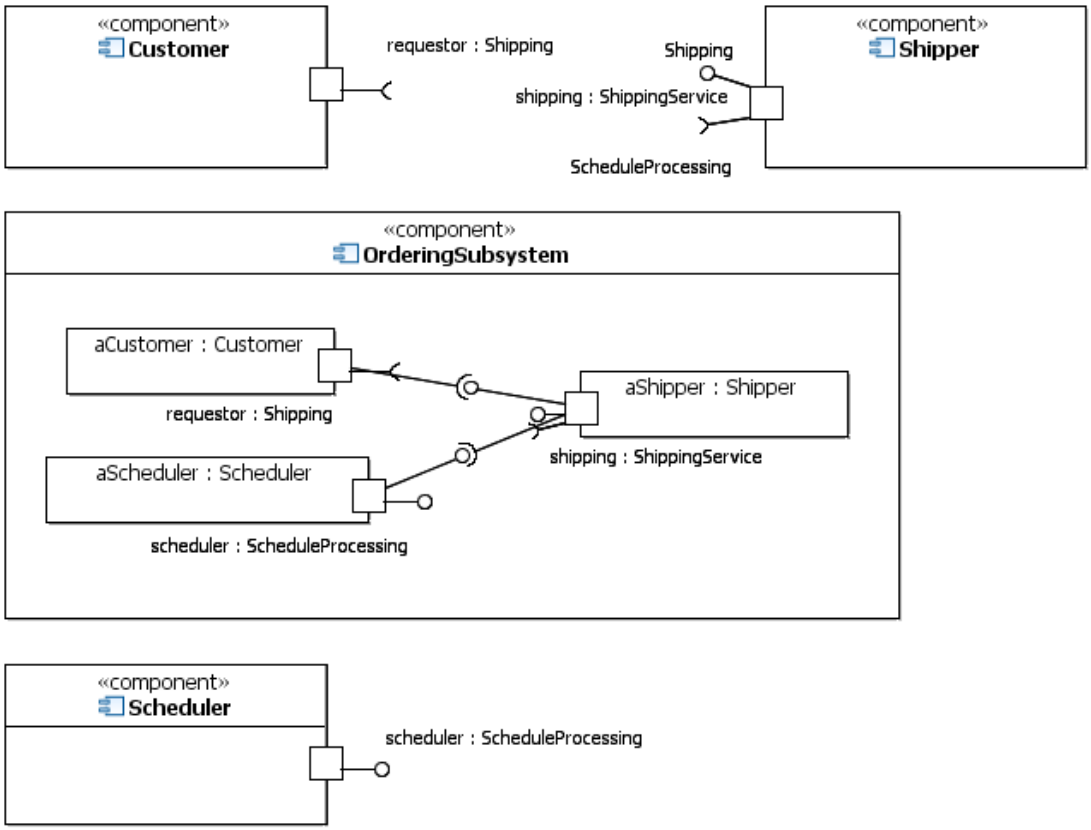

Figure 41: Using the shipping Service with two Consumers

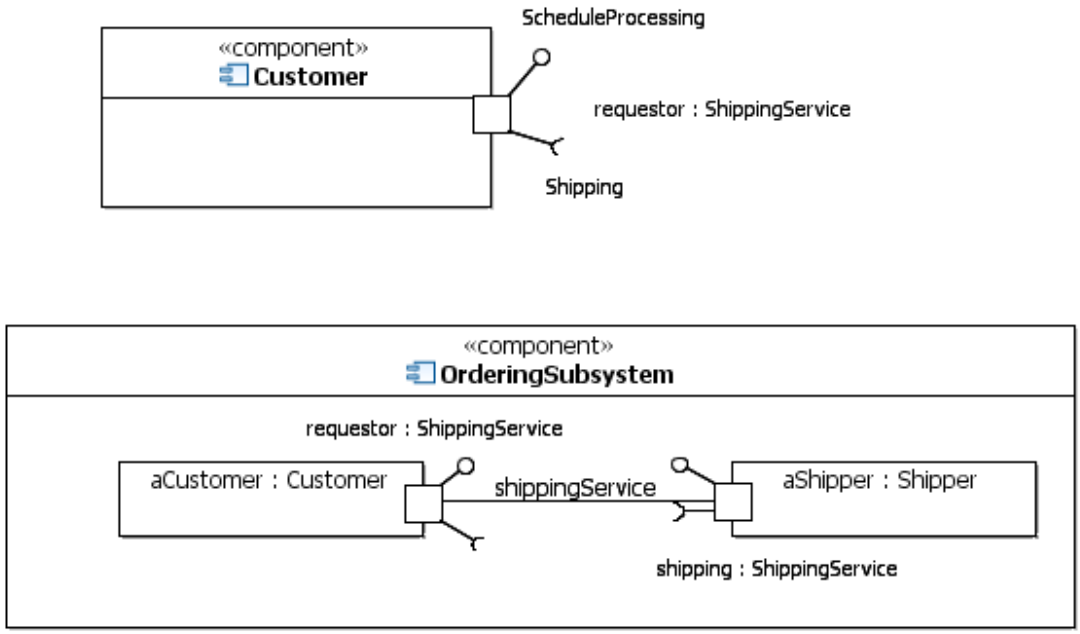

Figure 42: Using the shipping Service with one Consumer

Additions to UML 2.x

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

ServicesArchitecture

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

Extends Metaclass

- Collaboration or Component (Participant)

Description

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

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

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

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. Each participant satisfying roles in a ServicesArchitecture shall have a port for each role binding attached to that participant. This port shall have a type *compliant* with the type of the role used in the ServiceContract..

Semantics

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

Examples

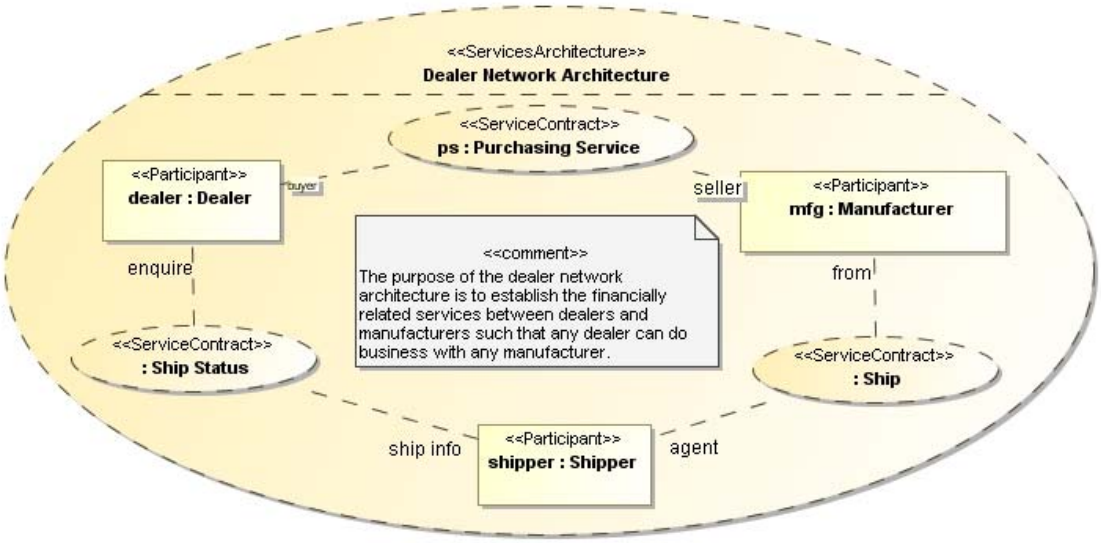

Figure 43: Services architecture involving three participants

The example (Figure 43) 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.

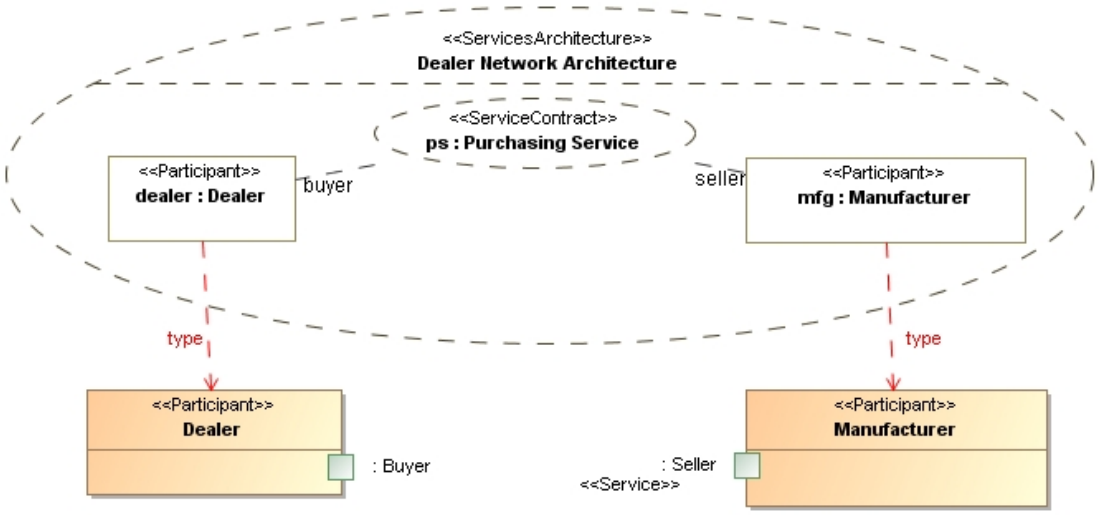

Figure 44: Abbreviated service contract

The diagram in Figure 44 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.

Participant Services Architecture

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

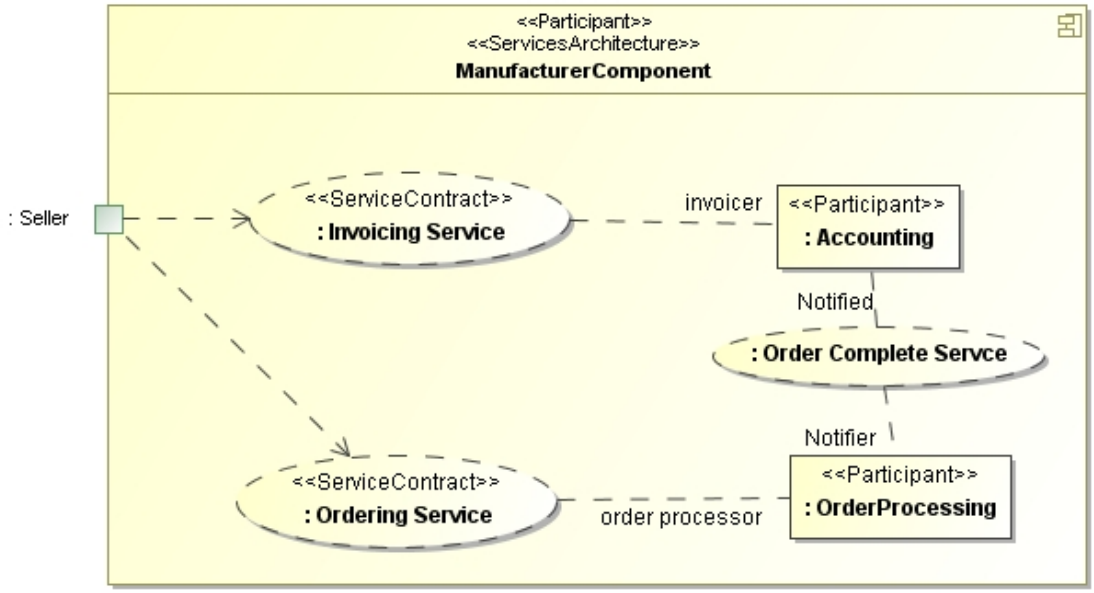

Figure 45: Participant's services architecture

Figure 45 shows a participant's services architecture. The "Manufacturer component" is composed of "Accounting" and "Order Processing". The "seller" service port on the Manufacturer component shows the external responsibility of the manufacturer which is then delegated to the accounting and order processing parts. In participant architecture there are frequently services connected between internal roles or between internal roles and external ports. The "Order CompppleteService" shows a service that is internal to the Manufacturer while both the "InvoicingService" and "Ordering Service" are delegated from the Manufacturer component to the internal participants, accounting and OrderProciessing, respectively.

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

ValueObject

Extends

- Cass
- DataType

Description

Represents a UML Class that does not have implicit, or system supplied identity.

Attributes

No additional attributes.

Associations

No additional Associations.

Constraints

No additional constraints.

Semantics

A ValueObject is a Classifier that has no identity, equality is determined by value, not by reference, and it can be freely interchanged between participants without regard to location, address space, local of control, or any other execution environment capability. Users should expect they are always working with a copy of any ValueObject.

ValueObject is essentially equivalent to DataType in UML2. However it is included in SoaML because of the common practice of using Class to represent data transfer objects rather than DataTypes. ValueObject supports this common practice while clearly distinguishing the role it plays in modeling.

Notation

A ValueObject is denoted as a Class with the «valueObject» Stereotype applied.

Additions to UML 2.x

ValueObject is included in SoaML to clearly distinguish objects with identity from those that do not. It is essentially the same as DataType in UML2, but is included because of the common practice of using Class instead of DataType for this purpose.

Classification

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, controler, 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.

Abstract Syntax

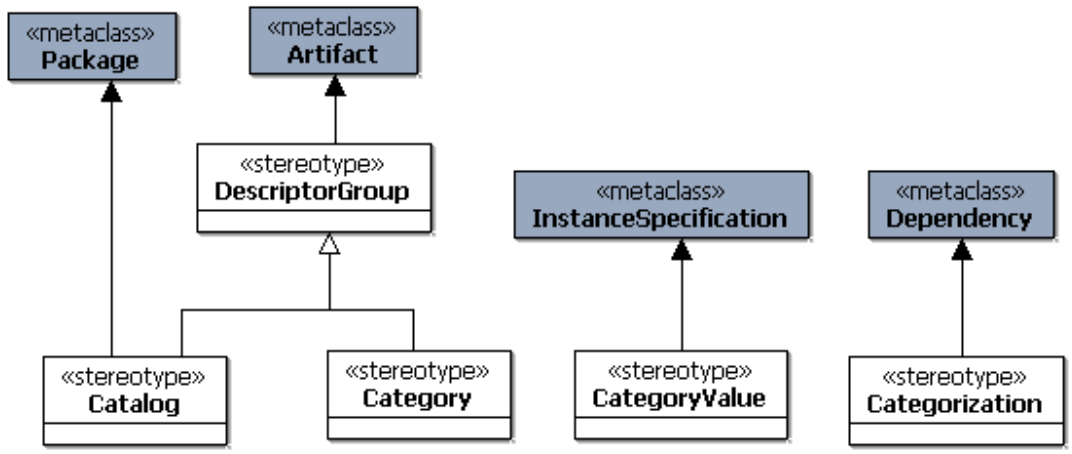

Figure 46: Classification

Class Descriptions

Catalog

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

Extends

- Package

Specializes

- DescriptorGroup

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 submission. 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.

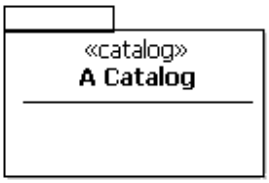

Figure 47: Catalog Notation

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 Categoization which may be represented as a Dependency with the «categorization» stereotype.

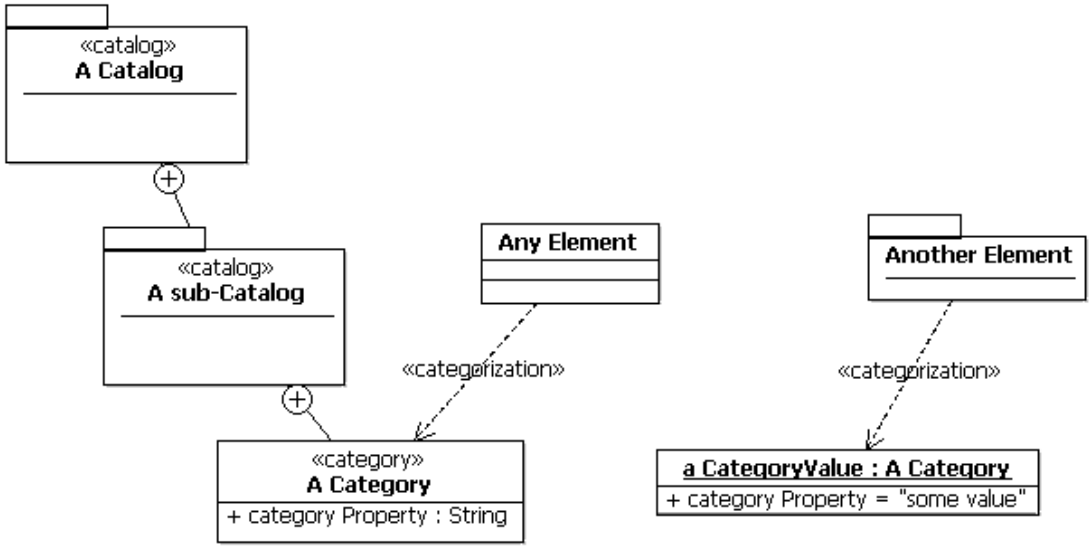

Figure 48: Notation to apply a Category or CategoryValue to a model element

Changes to UML 2.1

No changes to UML 2.1.

Category

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

Generalizations

- DescriptorGroup

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 which 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 49: Category Notation

Examples

Ownership

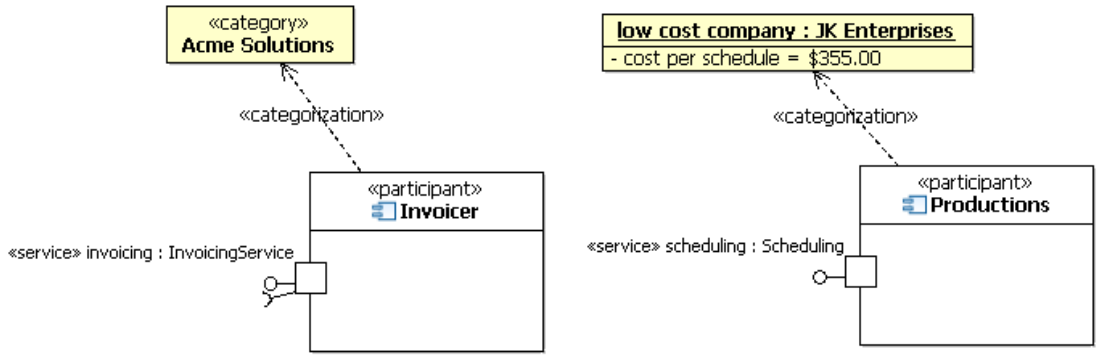

Figure 50: Ownership

Figure 50 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.

CategoryValue

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

Generalizations

- DescriptorGroup

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».

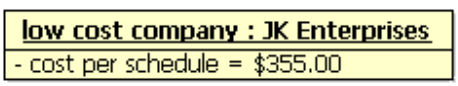

Figure 51: **CategoryValue Notation**

DescriptorGroup (placeholder for RAS DescriptorGroup)

A container for the Catalogs and Categories that categorize a model element.

Extends

- Artifact

Description

A DescriptorGroup is RAS element that is a container for a related group of descroptors.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

No additional constraints.

Semantics

DescriptorGroup provides a link between Catelogs and Categories used to categorize model elements, and RAS classification. A Catalog is a RAS DescriptorGroup containing DescriptorGroups representing other Catalogs and/or Categories.

Additions to UML 2.x

DescriptorGroup links Catalogs in SOA-ML to classifications in RAS.

BMM Integration

Overview

The promise of SOA is to provide an architecture of 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 assumes BMM is provided either as a UML profile, or as a package that can be merged with SoaML.

Abstract Syntax

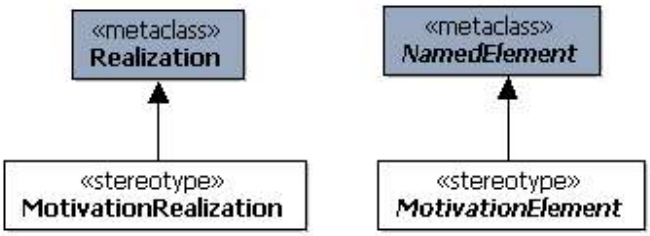

Figure 52: The BMM Integration Profile Elements

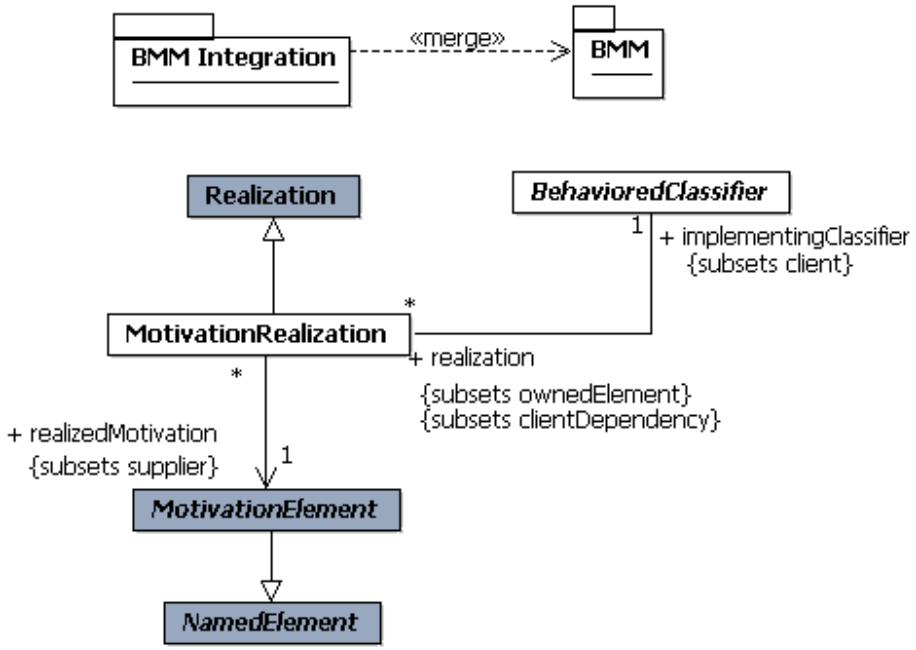

Figure 53: BMM Integration Package

Class and Stereotype Descriptions

MotivationElement

Generalizations

- NamedElement

Extensions

- NamedElement

Description

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

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 UML 2.x

No.

Examples

Figure 54 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.

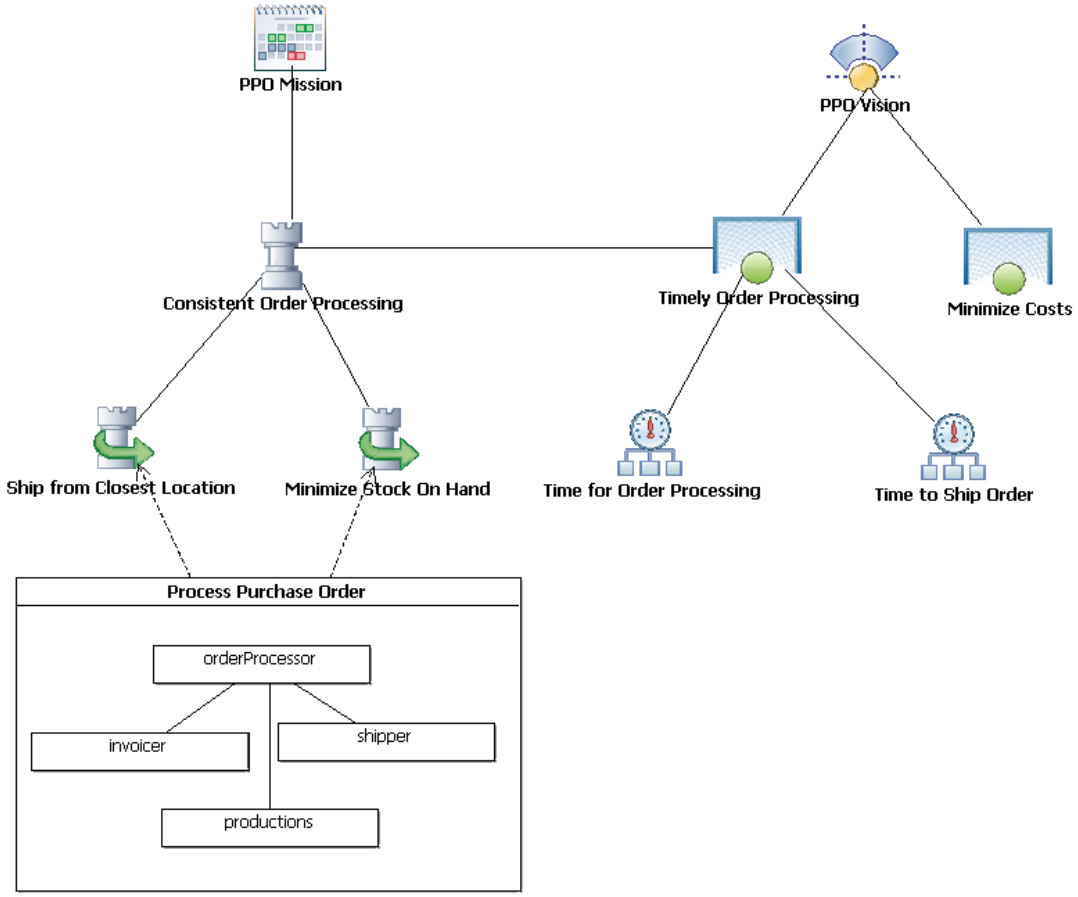

Figure 54: Business Motivation Model for Processing Purchase Orders

Additions to UML 2.x

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.

SoaML Metamodel

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, ServicesInterfaces, ServiceContracts and Service data. The participants allow 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 provide elements to model service messages explicitly and also to model message attachments.

Figure 55 the elements which 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 ports. 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.

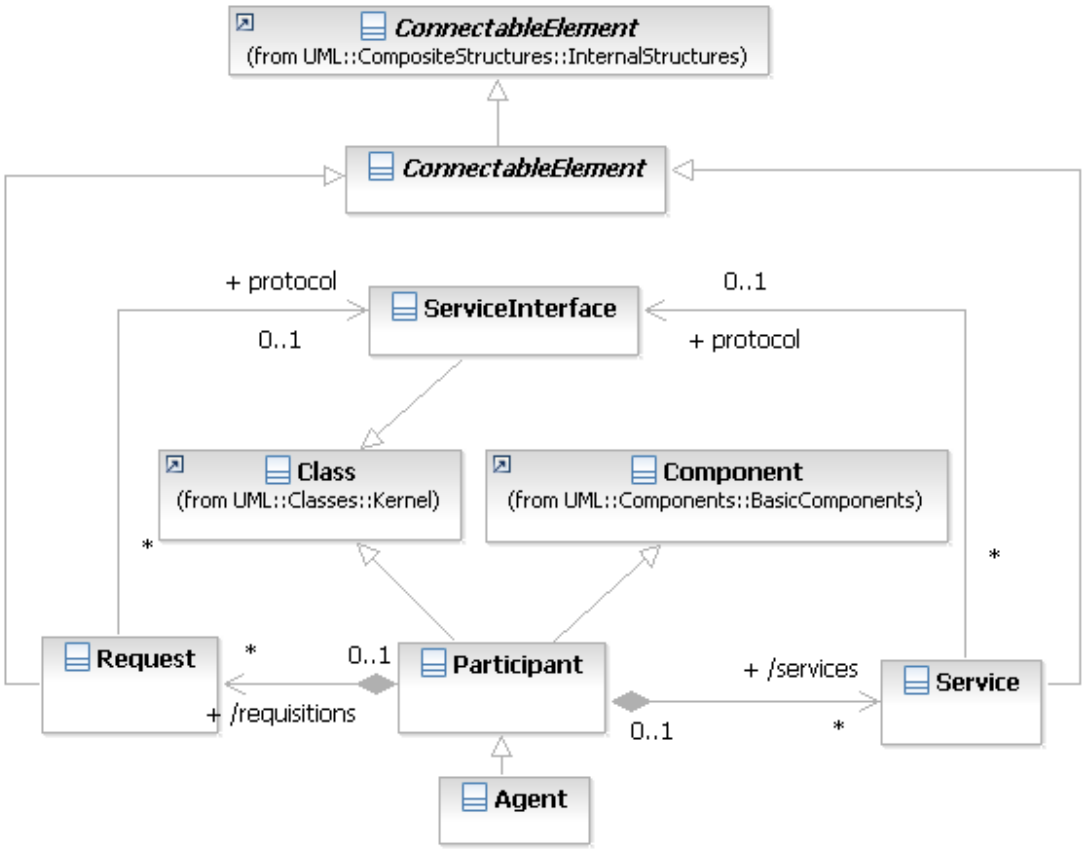

Figure 55: ServiceInterfaces and Participants

Figure 56 presents the elements which 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.

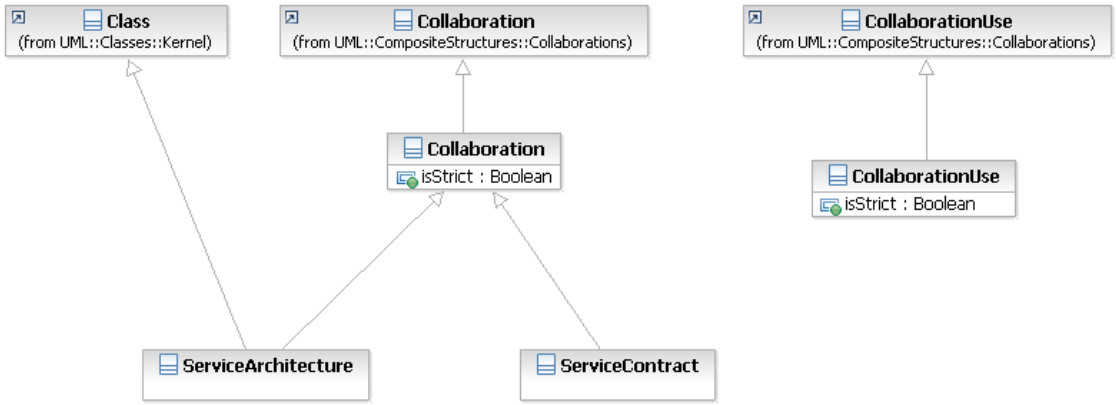

Figure 56: ServiceContracts and ServiceArchitectures

Figure 57 presents the elements which 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.

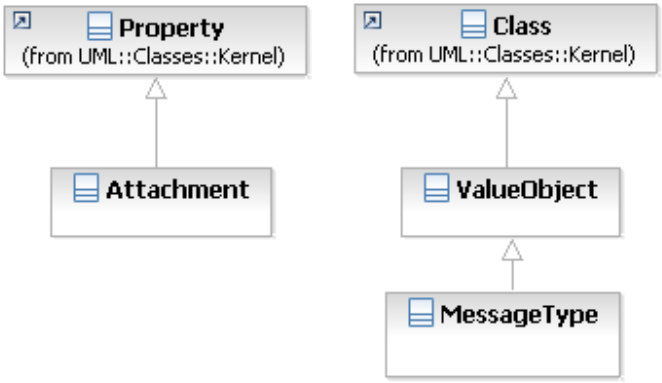

Figure 57: Data modeling

Class Descriptions

Agent

Figure 58: Agent

Definition

An Agent is a classification of autonomous entities that can adapt to and interact with their environment.

Description

An agent is a kind of Participant with some additional characteristics. Agents are autonomous, interactive and adaptive components. They are capable of acting without direct external intervention. They are able to communicate with the environment and other agents. They can learn and evolve over the time.

Attributes

No.

Semantics

See description.

Examples

No.

Attachment

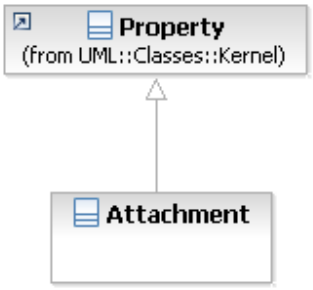

Figure 59: Attachment

Definition

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

Description

The objective of Attachment is to allow the definition of elements in the messages that have their own identity and which are separable from the main message. Examples of attachments can be images, word documents, pdf documents, or music samples.

Attachments are usually created by external applications and are expected to be processed by external applications.

Attributes

- encoding: <Primitive Type> String – It can be used to provide information on the codification of the attachment in the message

Semantics

An attachment is a specialization of Property. It helps to distinguish between regular attributes that usually have an associated datatype and attachments. Attachments are usually transmitted in binary format following a given encoding scheme. The encoding scheme can be defined in the encoding attribute.

Examples

No.

Collaboration

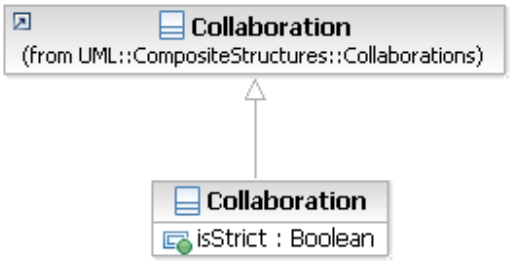

Figure 60: Collaboration

Definition

This is an auxiliary element to extend the UML collaboration with the isStrict attribute. This element is latterly extended by the ServiceContract and the ServiceArchitecture which are described in more detail. .

Description

No.

Attributes

- isStrict: <Primitive Type> Boolean – This attribute provides an indication of the level of adhesion to the behavior described in the Collaboration by the elements that play the roles contained in it.

Semantics

No.

Examples

No.

CollaborationUse

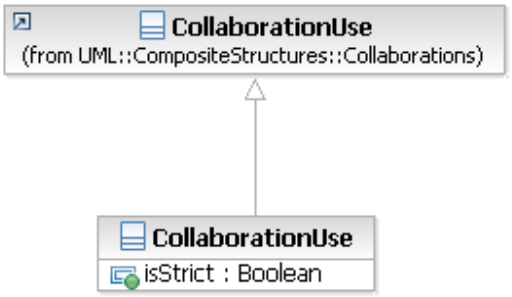

Figure 61: CollaborationUse

Definition

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

Description

The objective of a CollaborationUse is to show how a Collaboration (ServiceContracts and ServiceArchitectures) is fulfilled by the different inner parts of a structuredClassifiers. Particularly they are used within. ServiceContracts, ServiceArchitectures, Participants and ServiceInterfaces and are roleBinded to ConnectableElements (services and Request), Interfaces and Participants.

They can be used to show the roles that the different ConnectableElements of a Participant must comply with. They can also be used to show how ServiceContracts are build on top of other ServiceContracts. They can also be use to show the ServiceContracts implemented in a ServiceArchitecture.

Attributes

- isStrict: <Primitive Type> Boolean – This attribute provides an indication of the level of adhesion to the behavior described in the CollaborationUse by the elements that play the roles contained in it.

Semantics

When we link an element with a CollaborationUse (representing a ServiceContract or ServiceArchitecture) this implies that the linked element must satisfy the specification associated with the CollaborationUse. This can imply to implement operations, interfaces, and behaviors.

If not specified, isStrict is assumed to be true.

Examples

No.

ConnectableElement

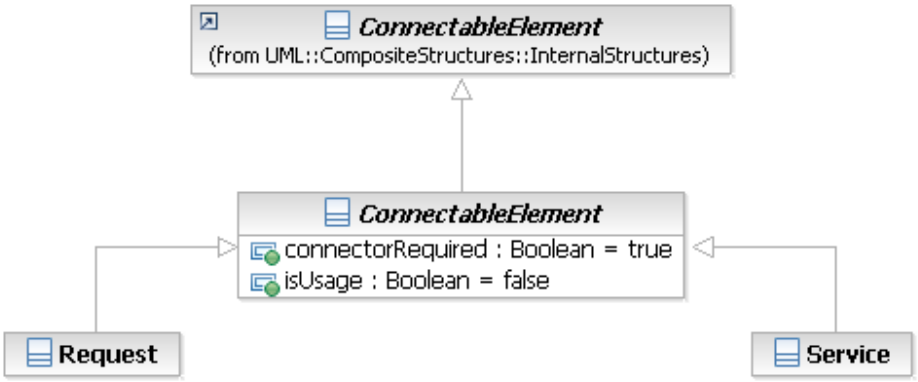

Figure 62: ConnectableElement

Definition

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

Description

Request and Service ports are both ConnectableElements from which information is exchanged with other external entities. The ConnectableElements are not always mandatory and it is possible to specify optional ConnectableElements using the connectorRequired attribute.

Participants can include one or more ConnectableElements to identify ports as service interaction points. The places from which the services are exchanges can be Services or Requests. Services are ConnectableElements that describe the services provided by the Participant, while Requests are ConnectableElements that describe the services that the Participant must consume in order to be able to provide their Services.

(Attr connectorRequired) ConnectableElement provide to Services and Requests with the connectorRequired attribute to indicate whether a connector is required on this connection point. This enables the designer to specify optional services and requests that are not critical for the participant.

(Attr isUsage) ConnectableElement also provides the isUsage attribute that allows to further distinguish between Services and Request. The default value for this attribute is false, but for Request the default value for these elements is switched to true.

Attributes

- connectorRequired: <Primitive Type> Boolean – If the value is true, there must be a connector attached to the ConnectableElement.
- isUsage: <Primitive Type> Boolean – The default value for this attribute is false, but for Request the default value for these elements is switched to true.

Semantics

(Attr connectorRequired) When a ConnectableElement, a Service or a request, has the attribute connectorRequired is set to true (the default value) this implies that there must be at least one connector to that ConnectableElement. This means that if the value is true the Service must be consumed or the request must be satisfied.

On the other hand, when the value of connectorRequired is set to false this implies that the Participant is able to work without that Service or request. This may have some implication on the service quality that the Participant should be able to manage.

(Attr isUsage) When the value of the isUsage (Requests) attribute of a ConnectableElement is set to true it means that the operation provided by the ServiceInterfaces are used and the Interfaces requested by the service interfaces are provided. If the value is false, it means that the provided by the ServiceInterfaces are provided and the Interfaces requested by the service interfaces are requested.

*(ServiceInterface) (R 6.5.15) (*Figure 63*)* The ConnectableElement inherits from UML TypedElement, therefore it has a type attribute. The type of the ConnectableElement determines the capabilities to be provided or requested through it. But the assignment of the type is optional. Therefore, if it is not assigned it means that the specification of the Service or Request may include or not the specification of the interfaces, operations and behaviours to by provided through the ConnectableElement.

(Interface/ServiceInterface) A ConnectableElement can be typed with interfaces. It is possible to use regular UML Interfaces or ServiceInterfaces defined in this metamodel. If UML Interfaces are used, it implies that the Participant that owns the ConnectableElement requires that capabilities from external entities, or provides capabilities to external entities. If ServiceInterfaces are used, it means that it will be possible to associate to that ConnectableElement one or more required and provided Interfaces between the provider and the consumer. The sequence of usage of operations of the different interfaces can be described by the ServiceInterface.

(Participant) When a ConnectableElement (Service or request) is attached to a Participant this means that the Participant is able to provide and process all the interface or the ServiceInterface associated to the ConnectableElement type. It also means that the Participant is able to cope with the roles of the Contracts to which the ConnectableElement is associated through a Role Binding *(*Figure 64*)*.

Examples

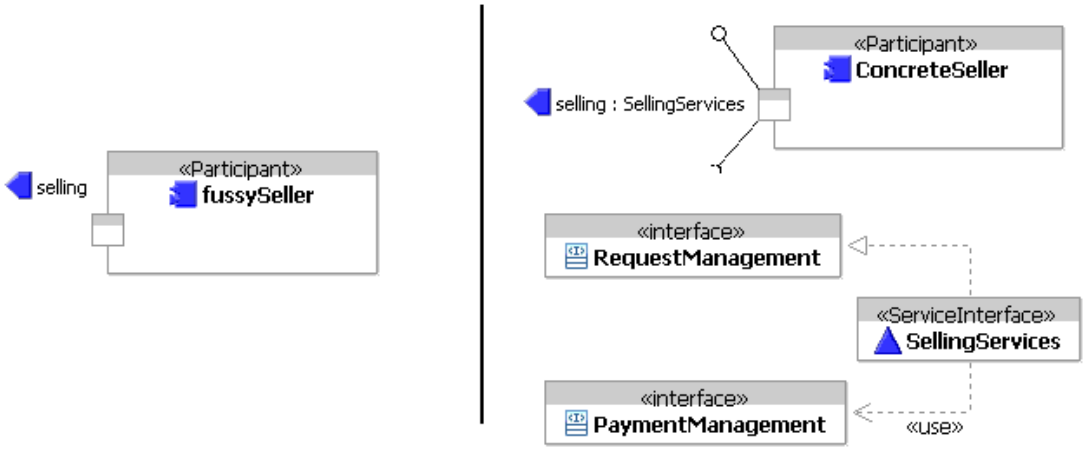

Figure 63: Diagram 01. OptionalServiceSpecifications

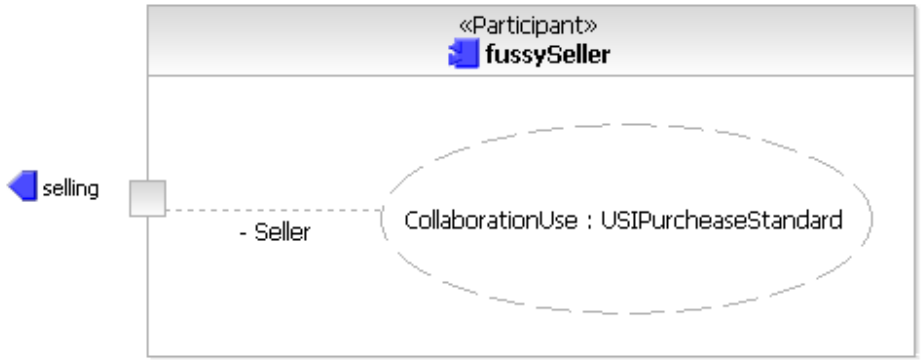

Figure 64: Diagram 02. RoleBindingContract

MessageType

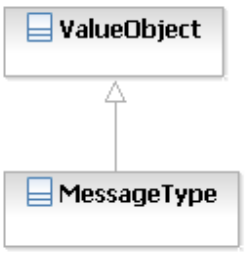

Figure 65: MessageType

Definition

The specification of information exchanged between service consumers and providers.

Description

A MessageType allows to explicitly identify the information to be exchanged through the services. Their usage is optional, as it is also possible to identify the messages exchanged from the signature of the operations of the ServiceInterfaces.

Their usage can be useful to identify those information structures that are especially sensible, and which cannot be changed without affecting the interaction of the Participant with other Participants.

MessageTypes can be used to describe input, output and exception messages.

Attributes

No.

Semantics

See description.

Examples

No.

Milestone

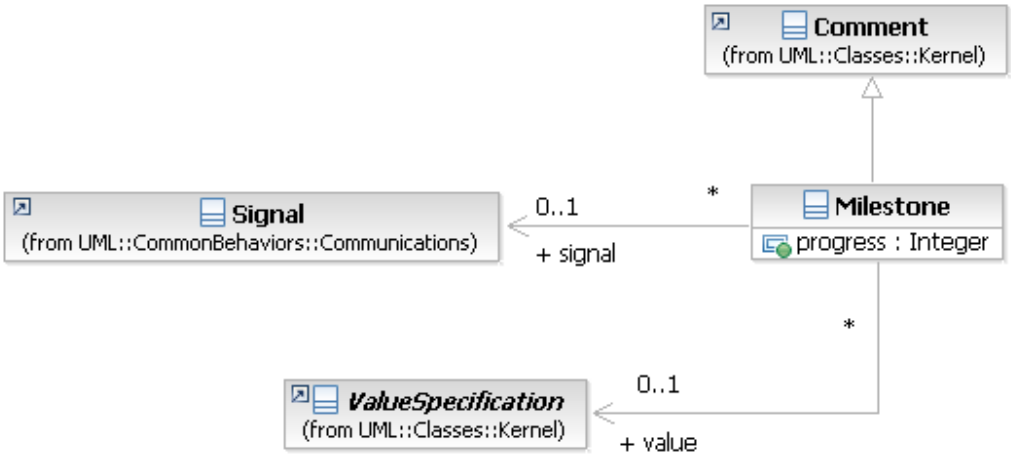

Figure 66: Milestone

Definition

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

Description

A milestone depicts progress by defining a signal that is sent to an imaginary 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.

Attributes

- progress: <Primitive Type> Integer – The progress measurement
- signal: <Class> Signal – Signal associated with the Milestone
- value: <Class> ValueSpecification – Value that arguments the signal when the Milestone is reached

Semantics

The milestones are used to annotate certain points in the behavioral descriptions of the services. They can be used in the behavioral description of the ServiceArchitectures, ServicesContracts, Participants and ServiceInterfaces.

The Milstone can be understood as a declaration to produce a signal at a specific point in a process. The signal is sent to an observer each time that a point connected to the Milestone is passed during execution. The implementation of the Milestones in the final execution environment is not mandatory, it is only advisable in those cases where there is an interest on monitoring the progress of the execution of certain behaviors.

Examples

No.

Participant

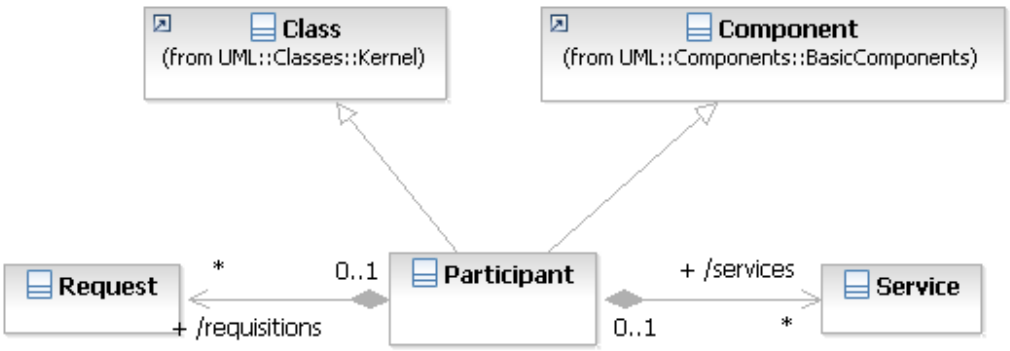

Figure 67: Participant

Definition

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.

Description

A Participant (Figure 68) is a special kind of UML Component or Class that provides or consumes services. It represents an element of the system, or the system itself, that interacts with other entities outside the boundaries of the system.

Participants are used to distinguish regular elements (Components and Classes) from those that maintain service relationships with entities outside the boundaries of the participant. For example they can be used in the design of a supply chain management system to differentiate those parts, or Components, that exchange information with other organizations. Participants can also be used to identify the different organizations taking part in complex business transactions.

Attributes

- requisitions: <Class> Request – Derived attribute that associates the Request ports contained by the Participant
- services: <Class> Service – Derived attribute that associates the Service ports contained by the Participant

Semantics

As a UML class or Component the participant represents a modular part of a system *(*Figure 68*)* that encapsulates its contents and whose manifestation is replaceable. In the same way that components can, the Participant can package other UML elements to support its definition and to organize the model *(*Figure 74*)*.

Participants may act as service consumer, providers or both. A participant will act as pure consumer if it only owns Requisitions and no Services *(R 6.5.12.1) (*Figure 70*)*. On the other hand it will act as a pure provider if it only owns Services and no Requisitions *(R 6.5.13.1) (*Figure 69*)*.

(Services/Requests) When a ConnectableElement (Service or Request) port is added to a Participant, this implies that the Participant is able to provide or consume the capabilities defined that ConnectableElement. The capabilities can be specified Interface or ServiceInterface type of the ConnectableElement, or by the Contracts supported by that ConnectableElement. It is important to remark that when the Participant Services a ServiceInterface (or regular UML Interface) that means that the Participant is able to provide the provided interfaces and consume the required interfaces (This is because the Participant is acting as the Provider of the ServiceInterface). The situation is different when a Participant Request a ServiceInterface. In that case that means that the Participant is able to consume the provided Interfaces and provide the required Interfaces (This is because the Participant is acting as the consumer of the ServiceInterface).

(ServiceInterfaces) The Interfaces or the ServiceInterfaces implemented by the Participant are referenced through ConnectableElements (Services or Requisitions). When a ServiceInterface is pointed through a ConnectableElement that means that the

Participant must implement the operations of the ServiceInterface and must comply with the behavior of the ServiceInterface.

(Contracts) Participants can indicate the service contracts they fulfill *(R.6.5.6.1)* and the roles they implement on those contracts. There are several ways to specify this relationship. It can be done in the context of a service architecture *(*Figure 71*)* or it can be done in the context of a participant *(*Figure 72*)*. The implication of this binding is that the participant should, or must *(R.6.5.6.2)* (in case the Contract is Strict), implement all the responsibilities specified in the contract for the linked role. For example, if the role type is a service interface, the participant should (or must) provide that service interface complying with the behavior of the contract. If the role type is a participant, this participant should (or must) provide all the features of that participant complying with the behavior of the contract.

(Behavior)(R6.5.14.1)(R6.5.14.1) Participant can include behavioral descriptions to describe the way in which the different Services and Requisitions are executed to complete the objective of the Participant. This specification does not constrain the way to describe the behavior of the component. Any UML behavioral specification method can be used to describe the activity of the Participant. This includes, but not is not limited to, activities, interactions, state machines, protocol state machines, and/or opaque behaviors *(*Figure 73)(Figure 76*)*.

Examples

Figure 68: Diagram 01. Simple

Figure 69: Diagram 02. Provider

Figure 70: Diagram 03. Consumer

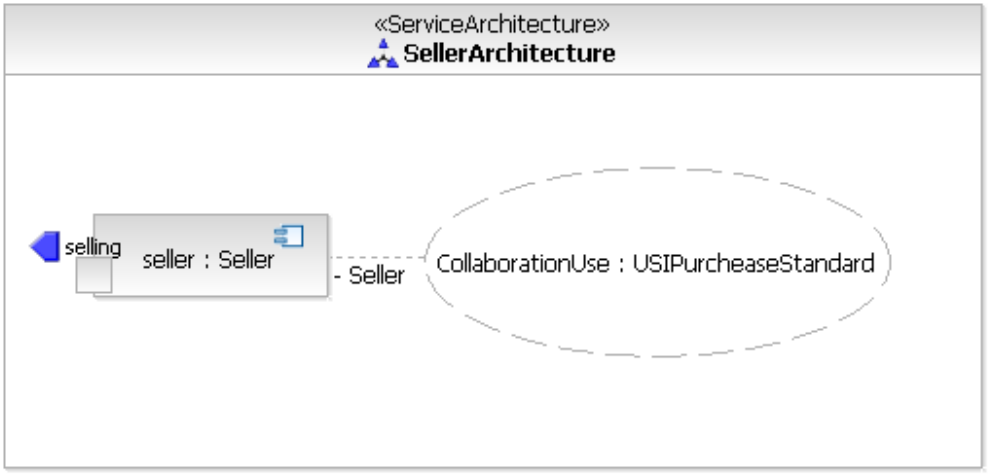

Figure 71: Diagram 04. ContractFulfillServiceArchitecture

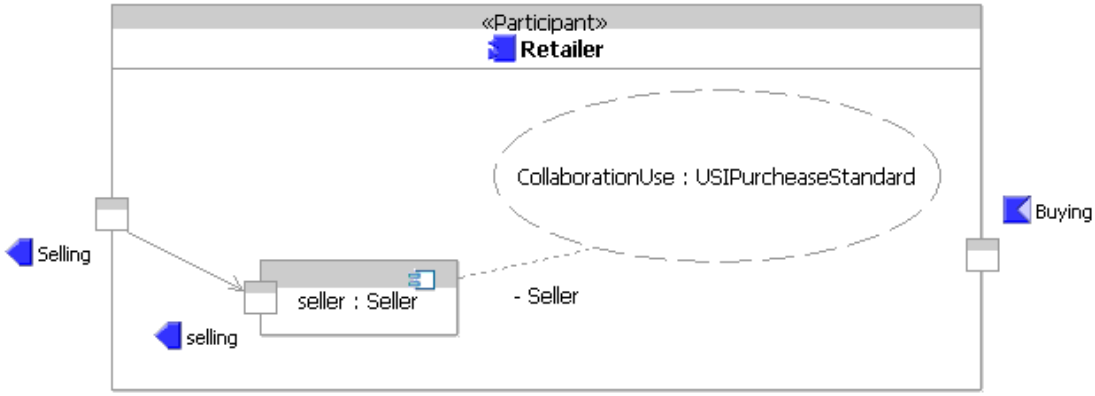

Figure 72: Diagram 05. ContractFulFillParticipant

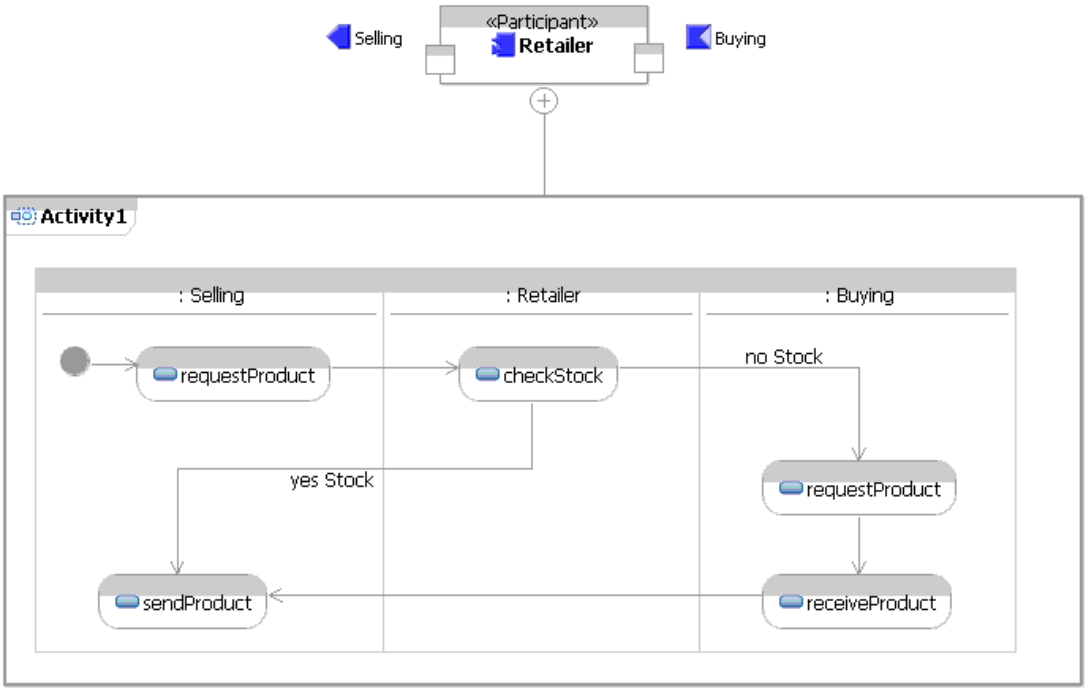

Figure 73: Diagram 06. Behavior

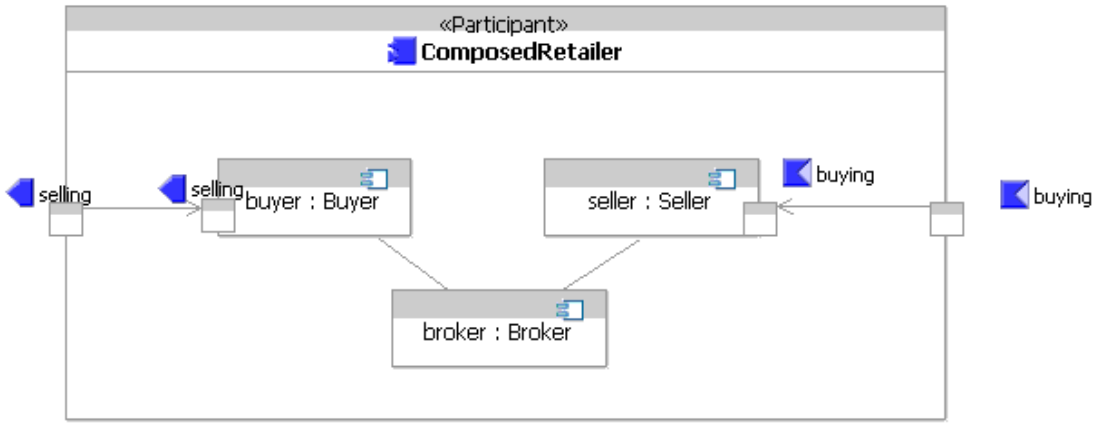

Figure 74: Diagram 07. ComposedParticipant

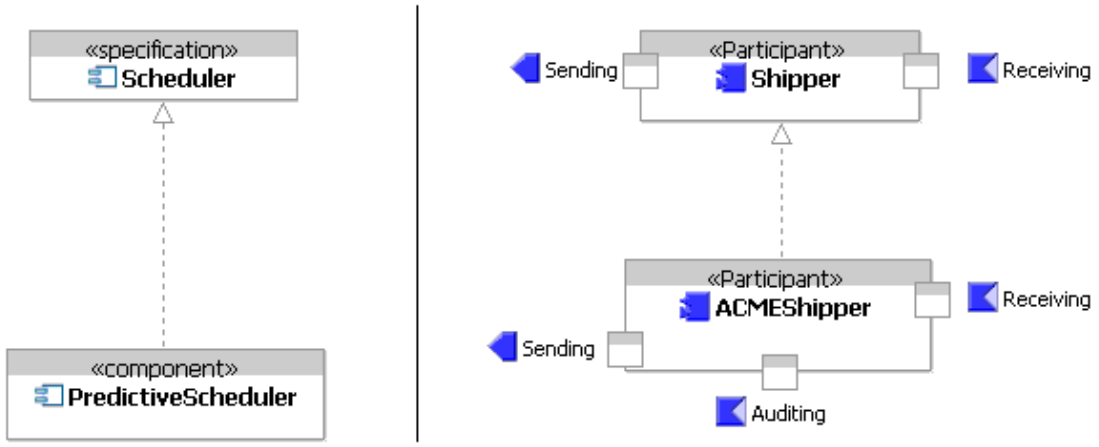

Figure 75: Diagram 08. SpecificationImplementation

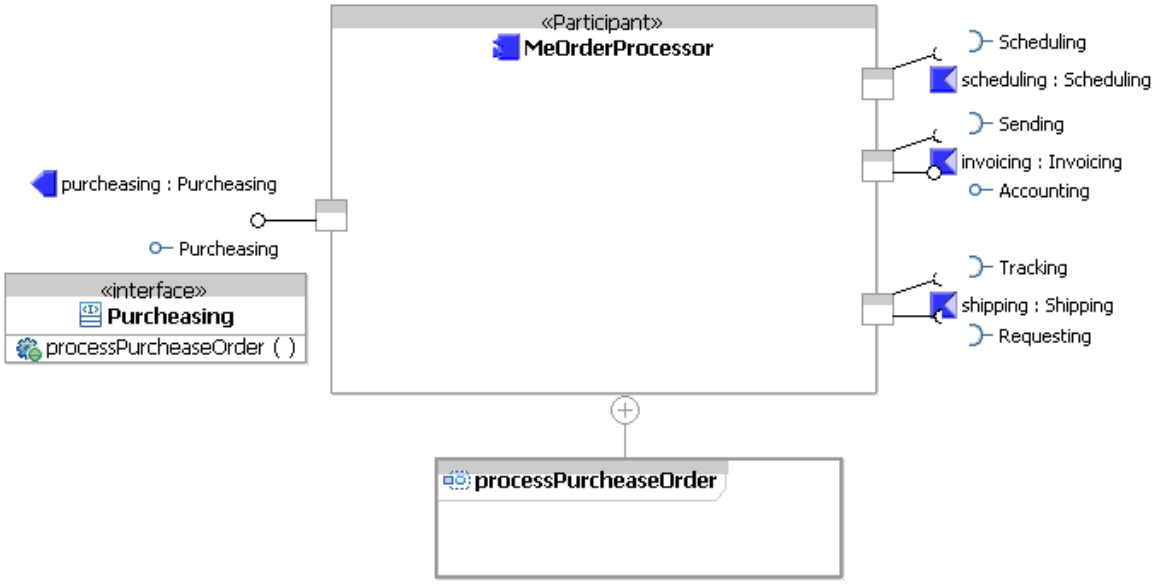

Figure 76: Diagram 09. BehaviorOperation

Property

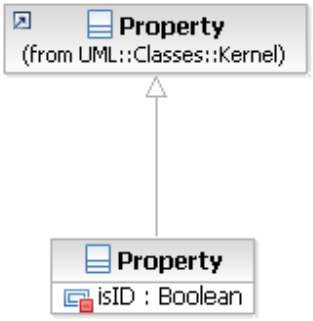

Figure 77: Property

Definition

The Property class 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”.

Description

The objective of this specialization of the UML2 Property element is to provide the mechanisms to identify instances of the containing classifier in distributed systems.

Attributes

- isID: <Primitive Type> Boolean – Indicates that the property can be used to identify the instances of the contained classifier.

Semantics

When the isID attribute is set to true that implies that then values assigned to that property should be unique in that system. These properties can be used to distinguish instances of the containing classifier.

Examples

No.

Request

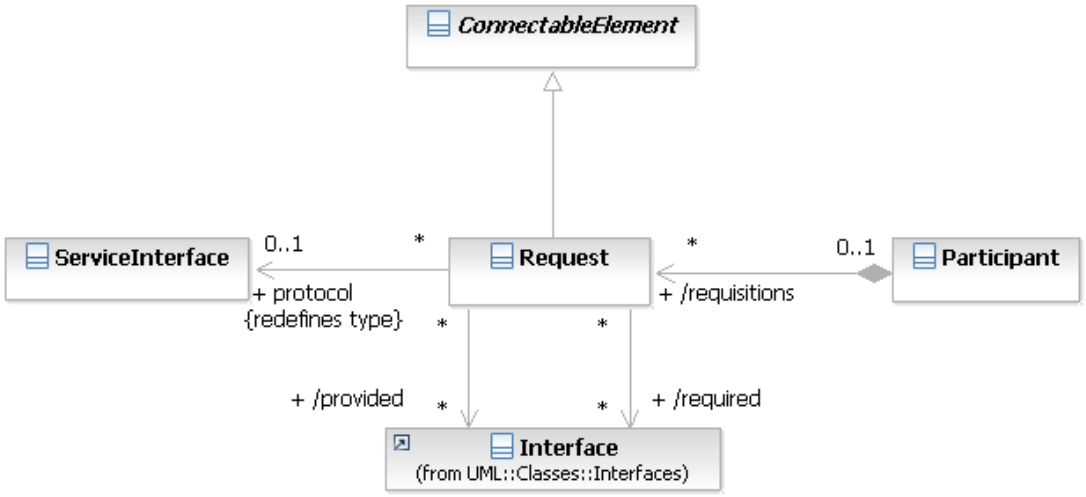

Figure 78: Request

Definition

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

Description

The objective of the Request port is to make it possible to distinguish the main service or services of the Participant and those who are required in order to be able to provide the main services. For example a Purchasing company main service is the PurchaseService, but in order to provide that service it may require other external services such as Invoicing, Scheduling or Shipping services.

Attributes

- protocol: <Class> ServiceInterface – Is used to associate the Interfaces that is implemented by this port
- required: <Class> Interface – Derived association to all the required Interfaces. The Interfaces are those implemented by the ServiceInterface of the protocol,
- provided: <Class> Interface – Derived association to all the provided Interfaces The Interfaces are those used by the ServiceInterface of the protocol,

Semantics

A Request inherits from ConnectableElement. It is a ConnectableElement that can provide and request operations to external entities. It redefines the isUsage attribute of the ConnectableElement to true. That means that the Participant who owns the request is action as the consumer of the services.

More concretely, if the type of the Request is an Interface, the Participant must be able to consume that interface. In that case the Interface will be required by the ConnectableElement. If the type of the Request is a ServiceInterface, it means that the Interfaces realized by the ServiceInterface will be required by the ConnectableElement, and the Interfaces used by the ServiceInterface will be provided by the ConnectableElement .It is important to note that this semantic differs from the standard UML semantics for port and its types.

Examples

No.

Service

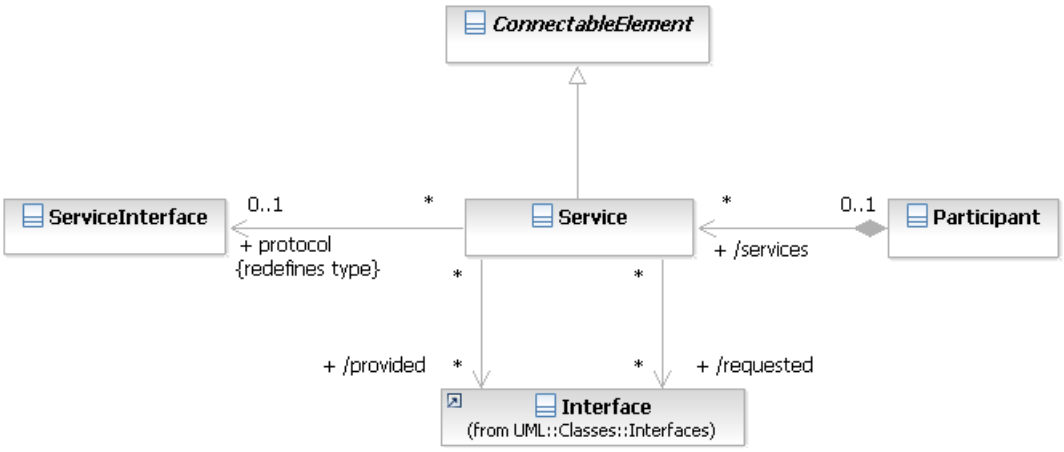

Figure 79: Service

Definition

A Service is a capability offered by one entity or entities to others using well defined terms, conditions and interfaces.

Description

The objective of the Service port is to make it possible to distinguish the main service or services of the Participant and those who are required in order to be able to provide the main services. For example a Purchasing company main service is the PurchaseService, but in order to provide that service it may require other external services such as Invoicing, Scheduling or Shipping services *(Fig Participant.Example.09.BehaviourOperation)*.

Attributes

- protocol: <Class> ServiceInterface – Is used to associate the Interfaces that is implemented by this port
- required: <Class> Interface – Derived association to all the required Interfaces. The Interfaces are those used by the ServiceInterface of the protocol,
- provided: <Class> Interface – Derived association to all the provided Interfaces The Interfaces are those implemented by the ServiceInterface of the protocol,

Semantics

A Service inherits from ConnectableElement. It is a ConnectableElement that can provide and request operations to external entities. From a conceptual point of view it identifies the interfaces that constitute the objective of the Participant.

More concretely, if the type of the Service is an Interface, the Participant must be able to provide that interface. In that case the Interface will be provided by the ConnectableElement. If the type of the Service is a ServiceInterface, it means that the

Interfaces realized by the ServiceInterface will be provided by the ConnectableElement, and the Interfaces used by the ServiceInterface will be required by the ConnectableElement.

Examples

No.

ServiceCapability

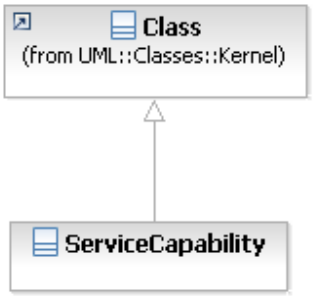

Figure 80: ServiceCapability

Definition

A ServiceCapability specifies a capability to provide a service.

Description

A ServiceCapability provides a simple way to discover and document a portfolio of services, including the capability to provide that service as internal behavior or the use of other service capabiliites.

ServiceCapability may specify its capabilities using Operations or realized Interfaces, and may contain ownedBehaviors indicating how the service capabilities might be implemented in order to discover and identify other services.

Attributes

No.

Semantics

A ServiceCapability may realize interfaces, or contain operations that list the necessary service capabilities. A ServiceCapability may also have ownedBehaviors which may be used to indicate how other ServiceCapability might be expected to be used to implement the service capabilities. These are primarily used to identify and document services.

A ServiceCapability is connected to ServiceInterfaces or Participants that realize it through a UML2 Realization. Like any other classifier, a ServiceCapabiltiy may be marked as a specification which may or may not be realized by some implementing ServiceCapability.

A ServiceCapability may also be used as a part in a Participant to which its services may be delegated for implementation purposes.

Examples

No.

ServiceContract

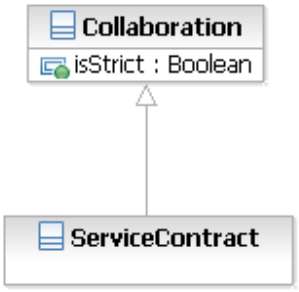

Figure 81: ServiceContract

Definition

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

Description

A ServiceContract formalizes the roles and responsibilities for providing and consuming a service, including the information, obligations and items that move between the participants in the realization of that service.

ServiceContracts are used to specify how a service ir provided and consumed based on interactions and behaviors involving multiple participants. At minimum, they specify the operations involved in the interaction and the sequence of invocation of those operations.

Participants and ConnectableElements can be bound to the different roles of the ServiceContracts. These bindings imply the capability of the Participant or the ConnectableElemenyt to fulfill that role.

Attributes

No.

Semantics

As a UML Collaboration the ServiceContract purpose is to describe the interaction among a set of roles. As a UML Collaboration the ServiceContract can also be used to realize UseCases that capture contract requirements *(R 6.5.5.1.c) (*Figure 84*).*

The role types in a ServiceContract are expected to be Interfaces or ServiceInterfaces *(. RolesTypes)*.

A ServiceContract can also nest in other ServiceContracts. This means that the realisation of the upper level ServiceContract forces the realization of the nested ServiceContracts.

Where the ServiceInterfaces are used as the types of parts in a service contract, the behavior of the serviceInterface must comply with the ServiceContract. However, common practice would be to specify a behavior in the ServiceContract or ServiceInterface not both.

(R6.5.5.1.f) UML Connectors can be used to indicate possible interactions between roles.

Examples

Figure 82: Diagram 01. Simple

Figure 83: Diagram 02. Roles

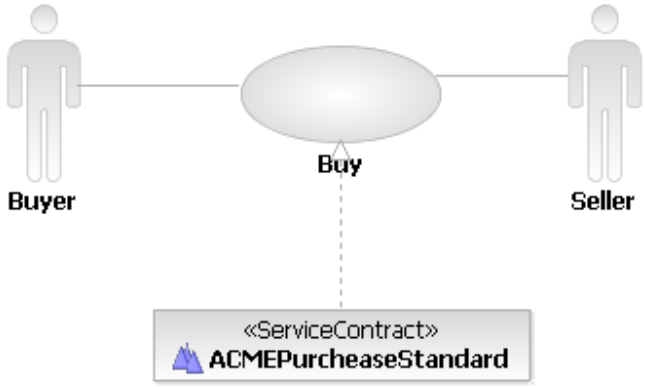

Figure 84: Diagram 03. UseCaseRealzation

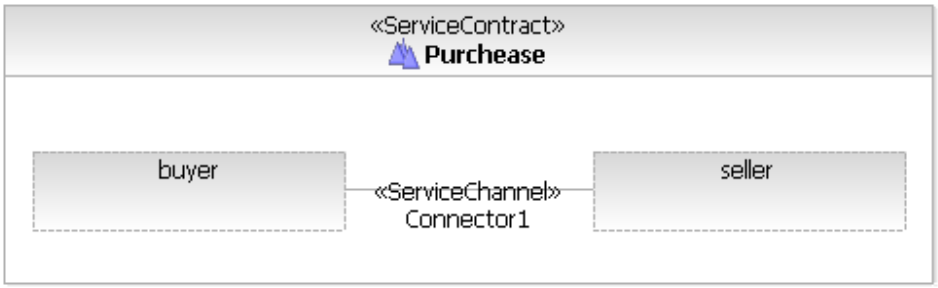

Figure 85: Diagram 04. Connectors

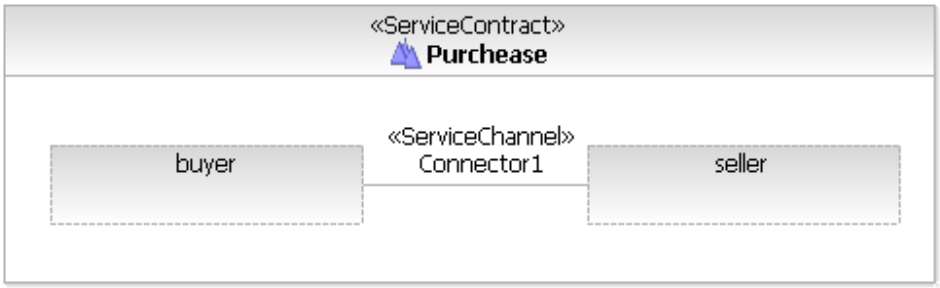

Figure 86: Diagram 05. ConstraintsAndObjectives

ServiceInterface

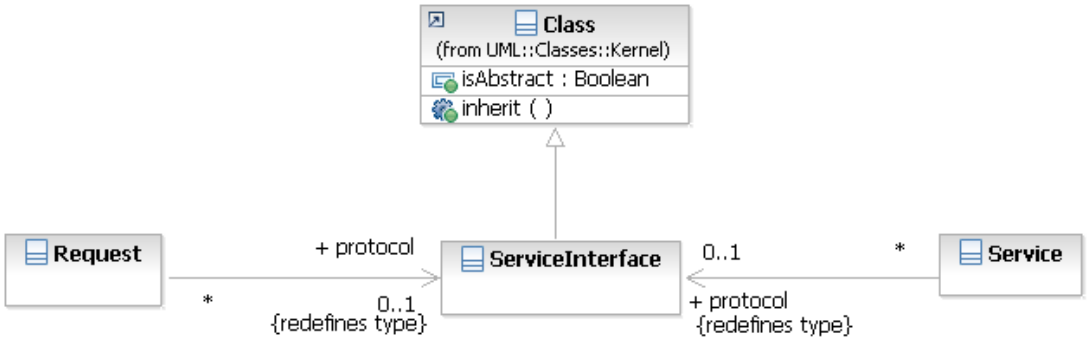

Figure 87: ServiceInterface

Definition

Defines the interface to a Service or Request.

Description

The ServiceInterface describe the operations used between a serviceProvide and a service Consumer from the perspective of the provider.

ServiceInterface are used to specify the operations involved in a Service interaction. It can also be used to specify the order in which those operations are executed. ServiceInterfaces are used as a type of a Service or Request. A ServiceInterface can

require the provision and requisition of operations, therefore a ServiceInterface can imply the implementation and usage of one or more regular UML Interfaces between the service provider and the service consumer.

Attributes

No.

Semantics

As a UML Class the ServiceInterfaces can realize and use other interfaces, this allows to represent complex services that require the bidirectional exchange of information between the provider and the consumer. The ServiceInterfaces are not intended to implement the behavior and the operations they define. ServiceInterfaces are used for specification purposes, the realization of the operations and the behavior is implemented by the concrete Participants.

ServiceInterfaces do not directly contain operations. The operations provided by the ServiceInterface are described in the operations of the realized interfaces. While the consumed interfaces are described in the operations of the used interfaces *(R 6.5.7.1.b)*. Using UML mechanisms operations can be further described with pre and post conditions, parameters and exceptions *(R 6.5.7.1.c)(R 6.5.7.1.d)*.

ServiceInterfaces can contain behavioral descriptions to describe the way in which the different provided and required operations are executed to fulfill the objectives of the service *(R 6.5.7.1.f)*.

Examples

No.

ServicesArchitecture

Figure 88: ServicesArchitecture

Definition

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

Description

ServicesArchitectures are used to conceptually relate sets of Participants that work together providing and consuming services for some purpose. ServicesArchitectures can be used to identify the Participants that work together with a common purpose and the Contracts that rule their interactions. ServicesArchitectures can be also use to group all the Participants that belong to a given domain or all the participants that belong to the same organization and relate there service relationships.

Attributes

No.

Semantics

The SevicesArchitecture is a kind of UML Collaboration or Class, as such, it can be further described with roles, parts and connectors. As a Collaboration it can be latterly used as CollaborationUse inside other architectures, to create grater or extended architectures over the existing ones.

ServicesArchitecture Inherits the isStrict attribute form the Collaboration, when the isStrict is false it means that is not mandatory to fulfill all the roles and behaviors specified by the ServicesArchitecture.

The Participants that collaborate in a service architecture are linked using Parts. The type of the Part Element identifies the Participant that the Part represents. It is also possible to include Contracts instances in the ServicesArchitecture. When a Contract is included in the ServicesArchitecture and it is related to a Participant, that implies that the Participant fulfills and is bound by that Contract (Figure 89).

Service contracts used inside the ServicesArchitecture can be linked to Participants or to the ConnectableElements (Services and Request) of the Participants. When a contract is linked to a ConnectableElements it means that the Participant fulfills that contract through that ConnectableElement (Figure 92).

Examples

Figure 89: Diagram 01. Participants

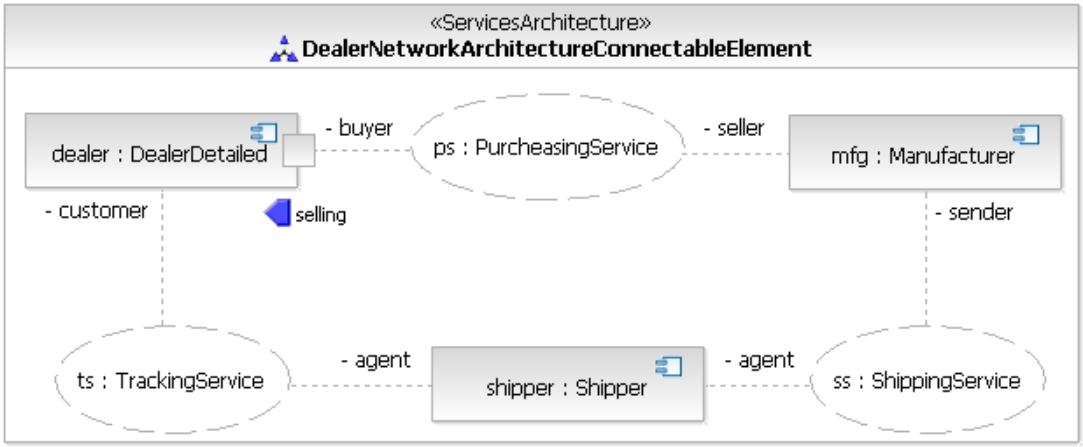

Figure 90: Diagram 02. ContractsParticipants

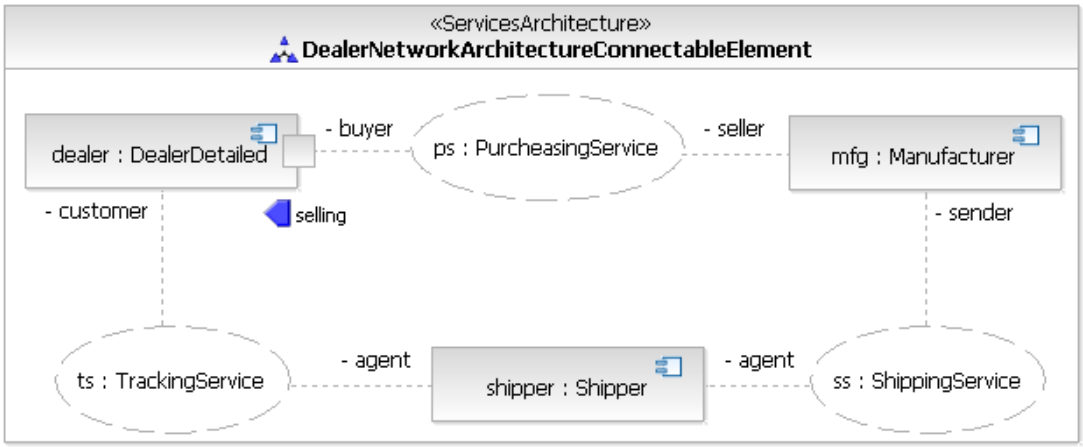

Figure 91: Diagram 03. ConnectableElement

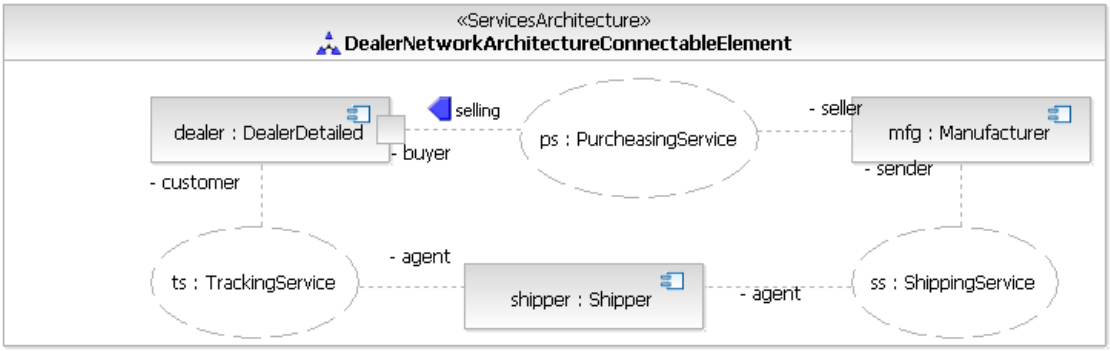

Figure 92: Diagram 04. ContractsConnectableElement

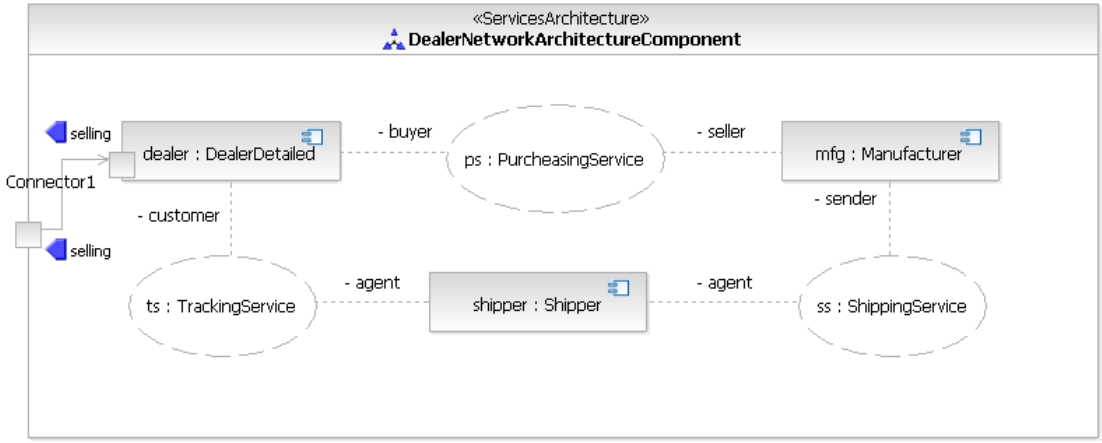

Figure 93: Diagram 05. ParticipantsComponent

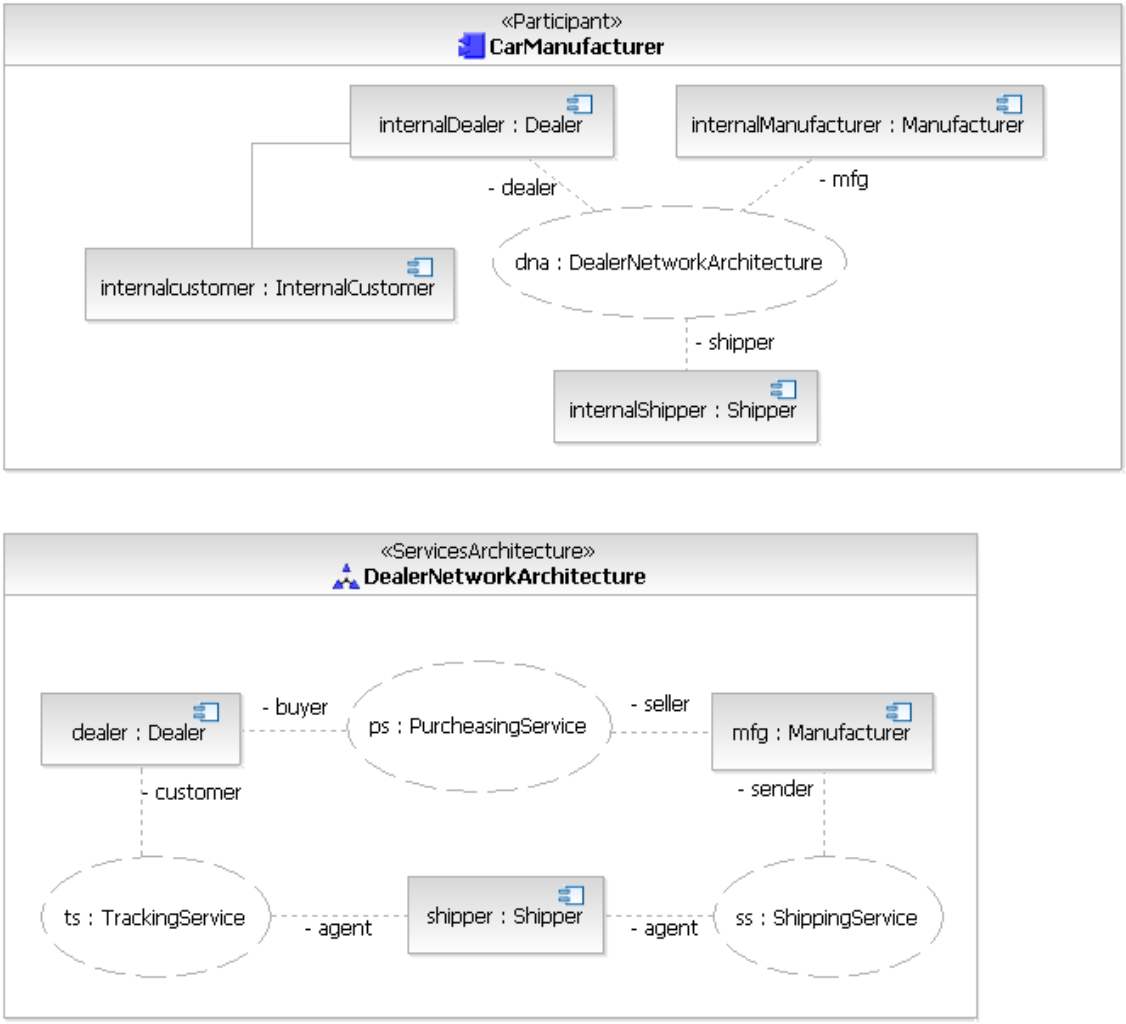

Figure 94: Diagram 06. ParticipantsCollaboration

ValueObject

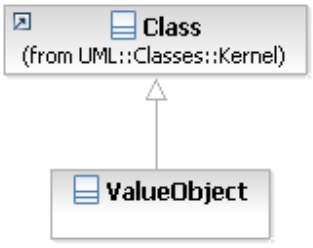

Figure 95: ValueObject

Definition

Represent a kind UML Class that does not have implicit, or system supplied identity. Their objects are always passed by value and not by reference.

Description

A ValueObject is a Classifier that has no identity. Equality is determined by value, not by reference, and it can be freely interchanged between participants without regards to location, address space, local of control, or any other execution environment capability users should expect they are always working with a copy of any ValueObject.

Attributes

No.

Semantics

ValueObject is essentially equivalent to DataType in UML2. However it is included because of the common practice of using Class to represent data transfer objects rather than DataTypes. ValueObject supports this common practice while clearly distinguish the role it plays in modeling.

Examples

No.

Profile metamodel mapping

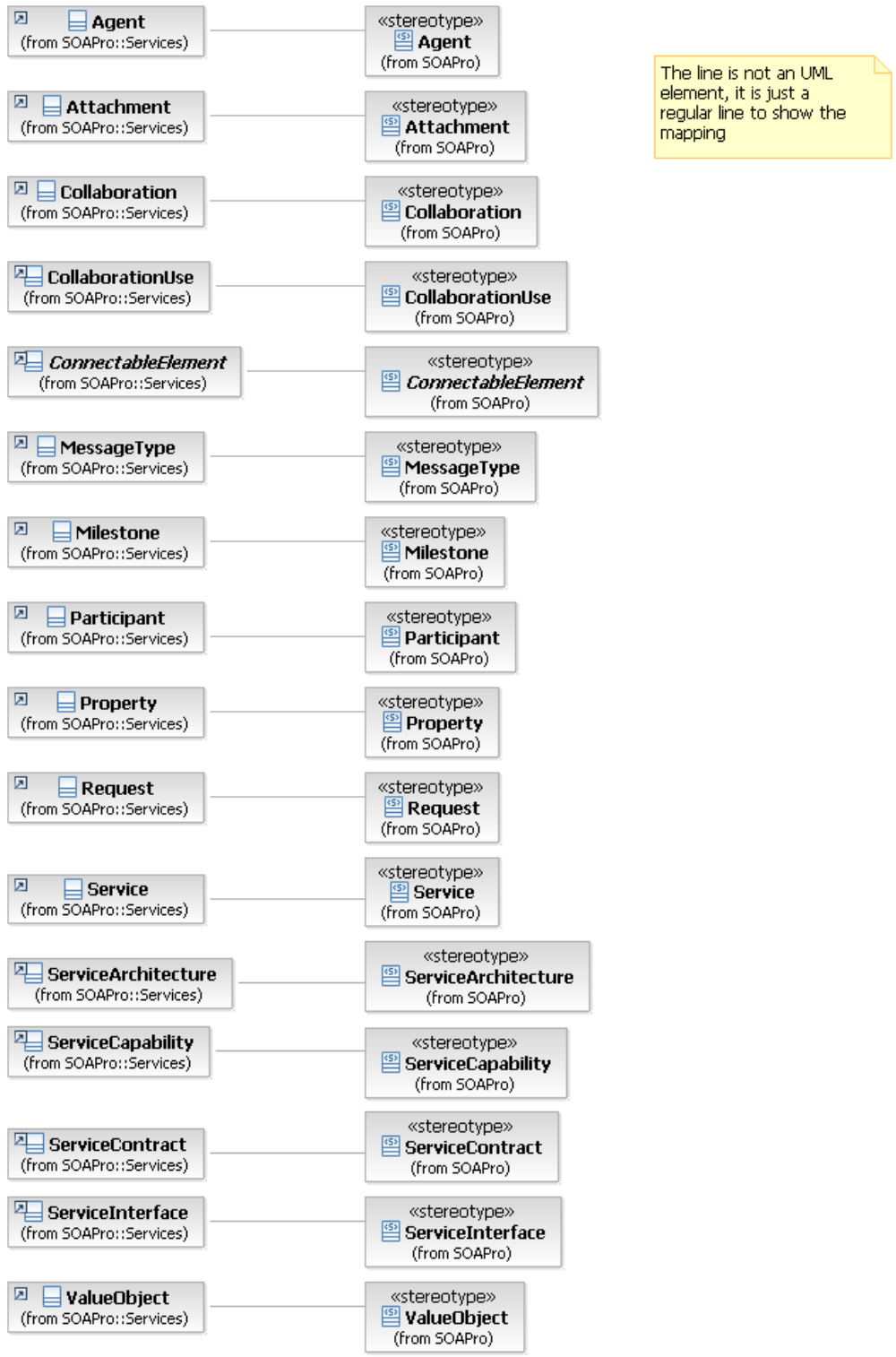

Figure 96: Profile metamodel mapping

Non-normative Appendices

Annex A: Sample Web Services Output

(non-normative)

Sample Web Services Output

Annex B: Conformance to OASIS Services Reference Model

This submission attempts to leverage existing work by OASIS and others to ensure conformance 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. Recently also the Open Group has published a draft SOA Ontology. (July 2008).

In the following 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 ServicesArchitetures.	*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			

	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
	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 the service, and is how it delivers value to its consumers.
Request (port stereotype)	A request models the use of a service by a participant and defines the connection point through which a Participant makes requests and uses or consumes services.		*Service Consumer:* A participant that interacts with a service in order to access a capability to address a need	
Service (port stereotype)	A capability offered by one entity or entities to others using well defined		*Service Provider:*	

	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
	terms, conditions and interfaces. The service stereotype of a port defines the connection point through which a Participant offers a service to clients.		A participant that offers a service that permits some capability to be used by other participants.	
Service (general) ServiceCapability	A capability offered by one entity or entities to others using well defined terms, conditions and interfaces. A service represents the ability of a participant to provide capabilities and value defined by a service interface that may be used to fulfill requests representing the desire of other participants to achieve needs and goals also defined by a service interface. A ServiceCapability specifies a capability to provide a service. A ServiceCapability models the capability for providing a service specified by a ServiceContract or ServiceInterface	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.
Service Contract	A ServiceContract is the formalization of a binding exchange of information, goods, or or obligations between parties defining a service. A ServiceContract is the specification of the agreement between providers and consumers	A **contract**, represents an agreement by two or more parties. A service contract is a measurable assertion that governs the requirements	*Adopts SOA-RM definition*	*Adopts SOA-RM definition*

	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
	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	and expectations of two or more parties.		
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,	An architectural paradigm for defining how people, organizations and systems	A paradigm for organizing	*Adopts SOA-RM definition*	An architectural style that supports service

	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
Services Architecture	provide and use services to achieve results. *Services Architecture* The high-level view of a Service Oriented Archtecture 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	and utilizing distributed capabilities that may be under the control of different ownership domains.		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 which 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 submission.

- *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 submission 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 submission 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; and*

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

- *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 submission.

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

Annex C: Examples

This section 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 submission. It grounds the concepts in reality, and shows the relationships between the parts. The example is elaborated in other sections in order to explain submission 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:

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

Dealer Network Architecture

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.

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.

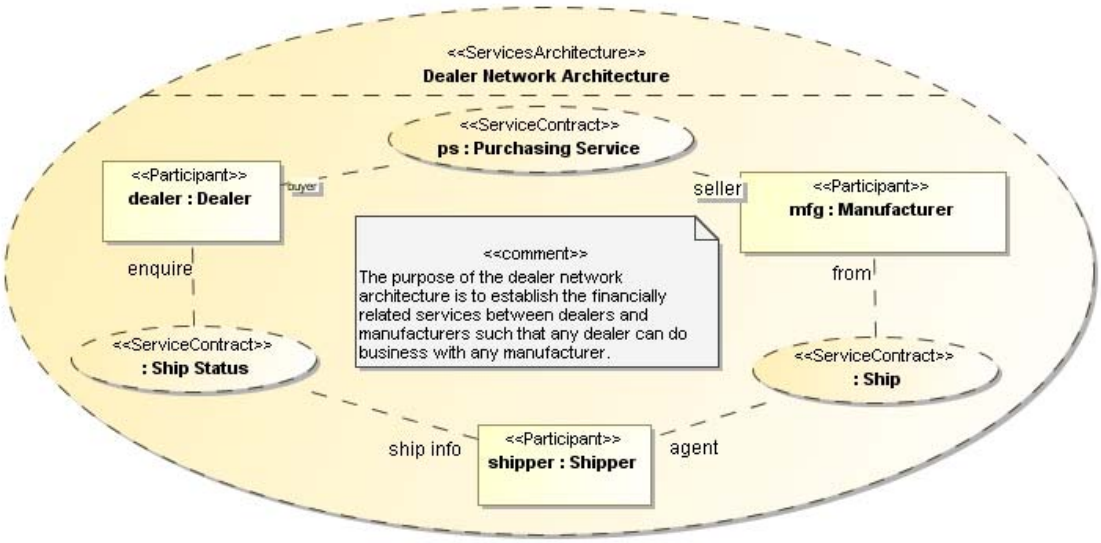

Figure 97: Dealer network architecture

Note that there is a direct correspondence between the roles and services in the 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.

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.

Services to support the community

Considering the "Purchasing Service"; the service in the Services Architecture does not define the service – it uses the service. 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 this diagram:

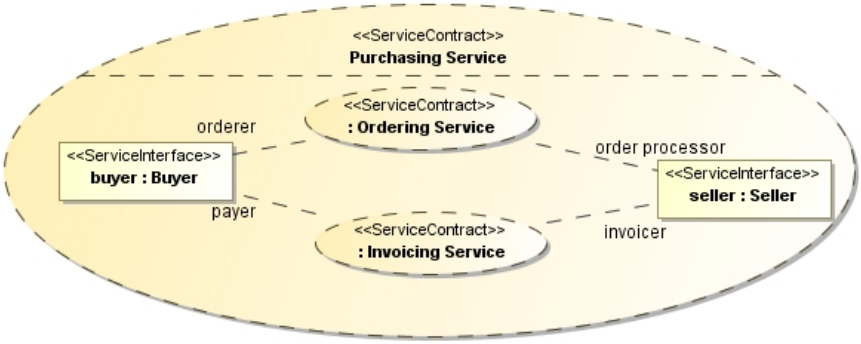

Figure 98: Purchasing service

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 like this:

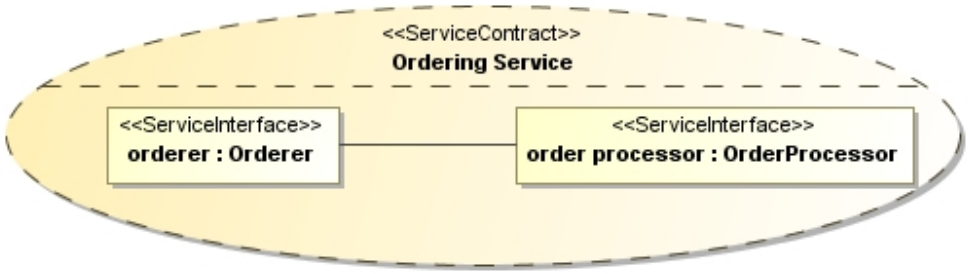

Figure 99: Ordering service

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 like this:

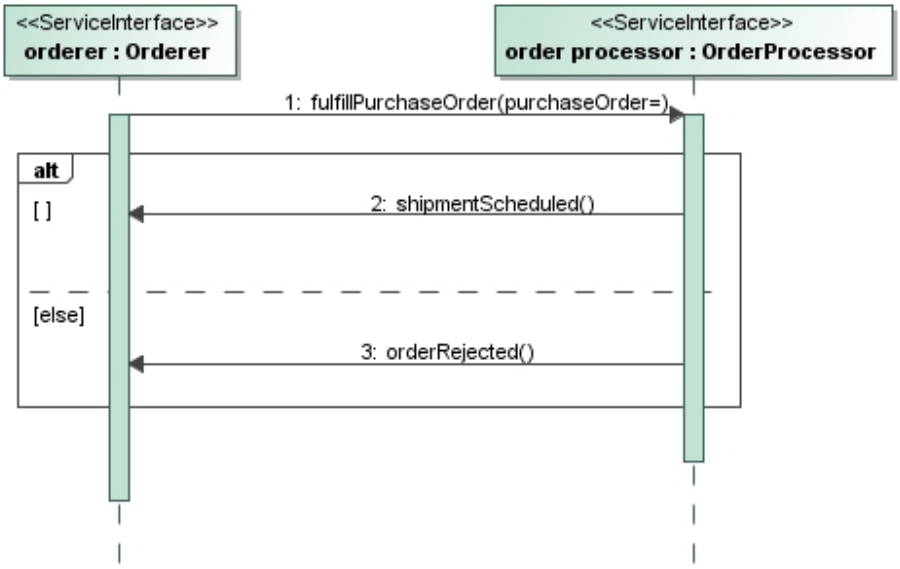

Figure 100: Flow of information

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.

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" the ther OrderProcessor and the orderProcessor sends back either a "shipmentSchedule" or an "oerderRejectied". 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.

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 there 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.

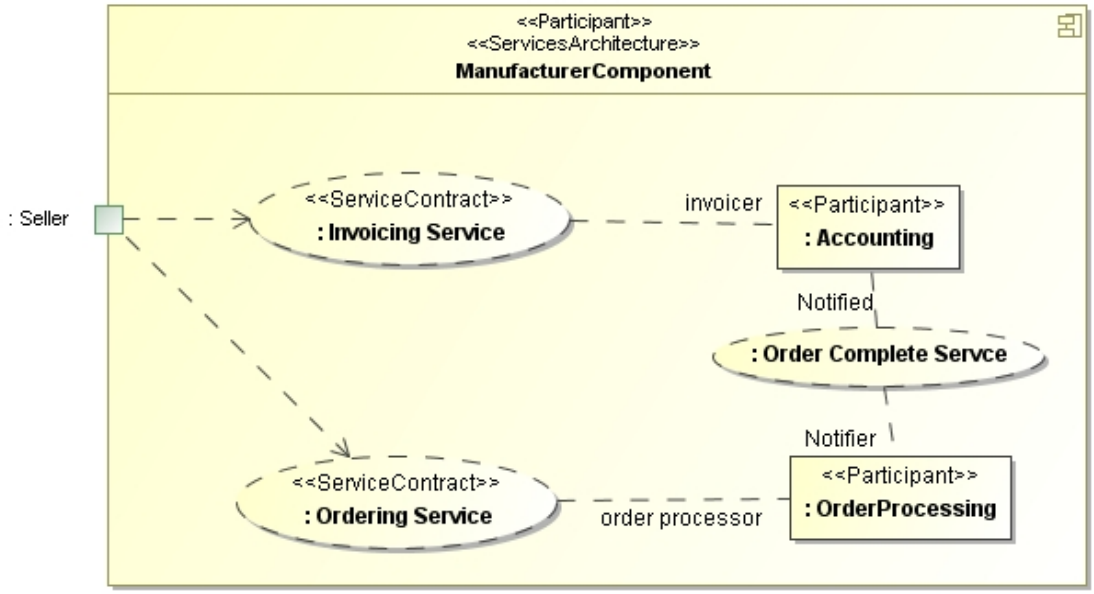

Figure 101: Manufacturer component

The diagram above shows the services architecture of a manufacture that complies with the community architecture. Note that in this case the community process 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, like this:

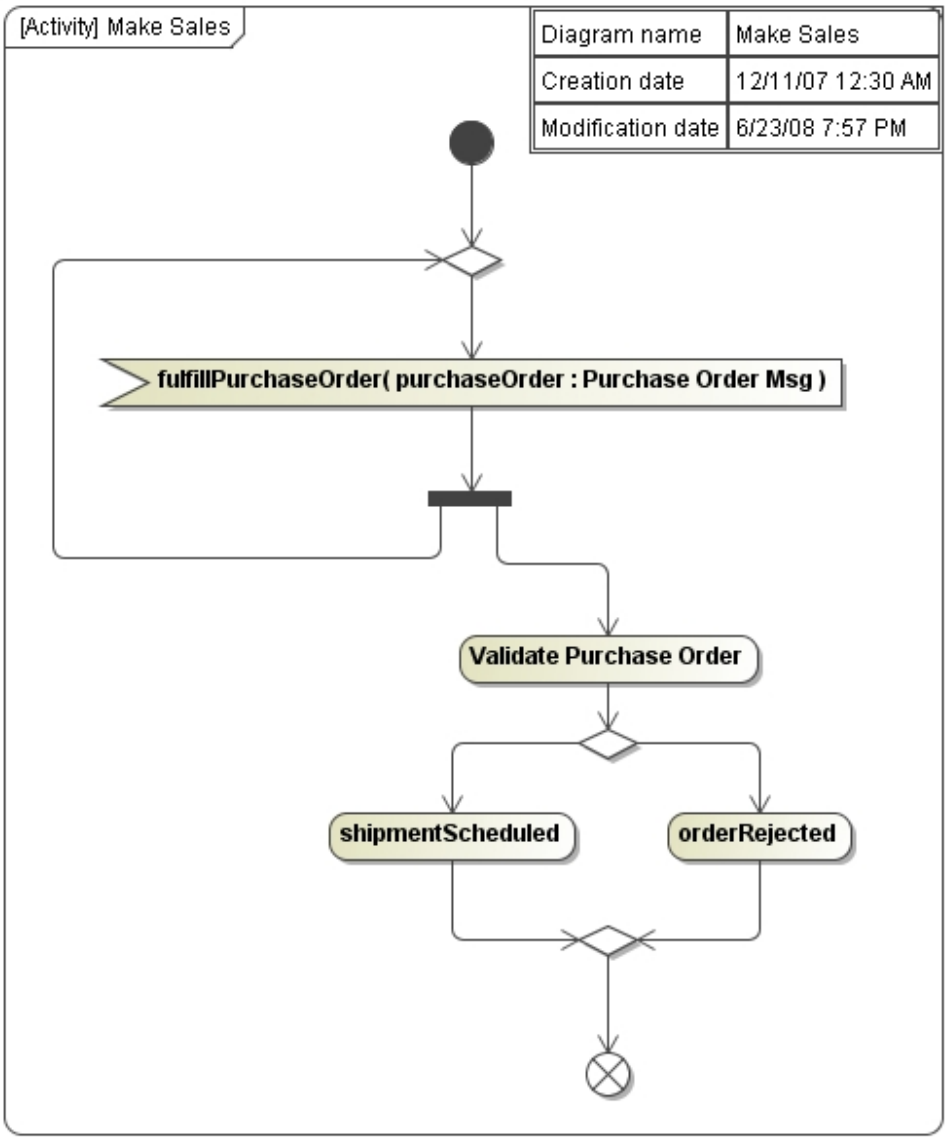

Figure 102: Make sales business process

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

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, we may want to define "components" that correspond to each of the roles and responsibilities we have defines. 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” which are the connection points where the services are actually provided or consumed.

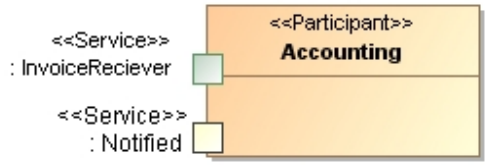

Figure 103: Accounting participant

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 “InvoiceReciever” 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 which 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).

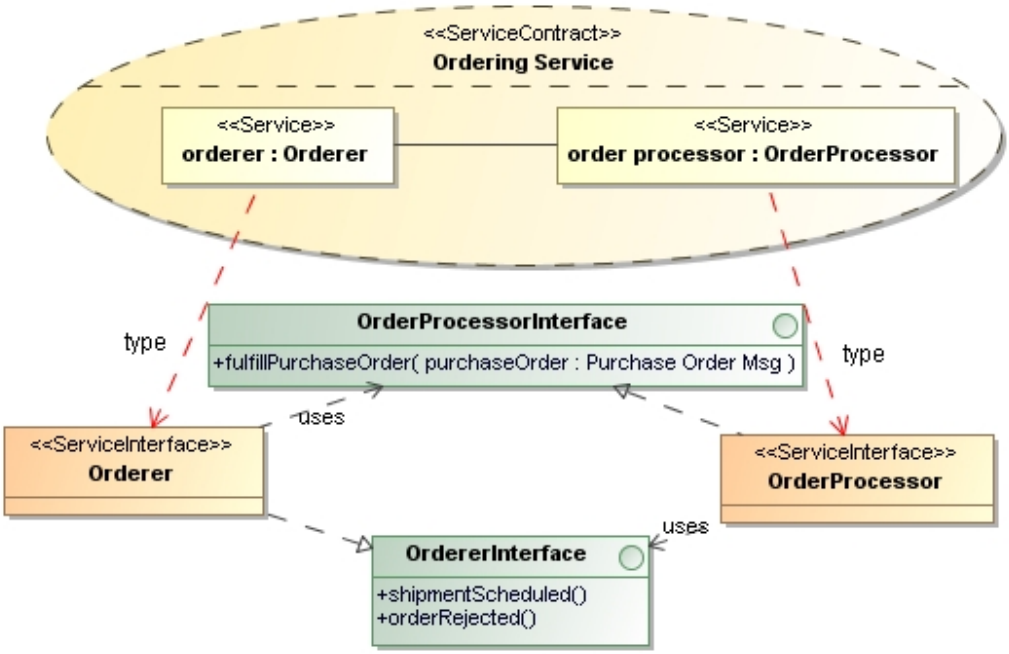

Figure 104: Service interfaces

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 as service contracts

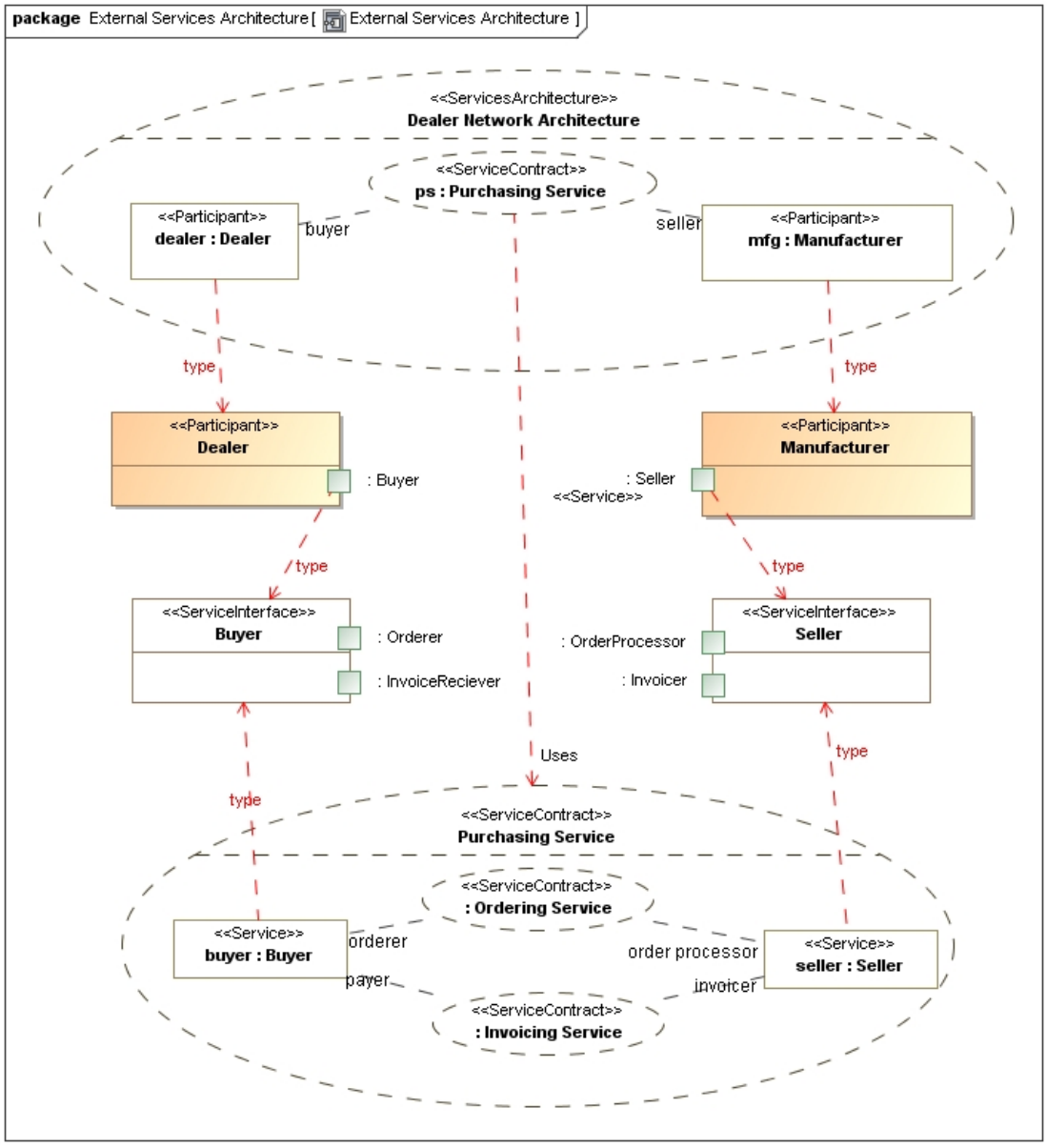

Figure 105: Relating services architecture as service contracts

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 2 ports because Purchasing Service is a compound service contract – the service interfaces have a port for each nested service: OrerService & 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 submission.

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. The collaboration is then fulfilled by a number of collaborating Participants having needs and capabilities through Requisitions and Services specified by ServiceInterfaces identified by examining the collaboration. SoaML supports a number of different ways to capture and formalize services requirements including ServicesArchitectures, ServiceCapability 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.

The Services Solution Model

Requirements

The requirements for how to processing purchase orders is 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 is treaded 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, nor any connections between the participants. The services architecture will eventually result in the connections between participants with requests representing needs and participants with services that satisfy those needs.

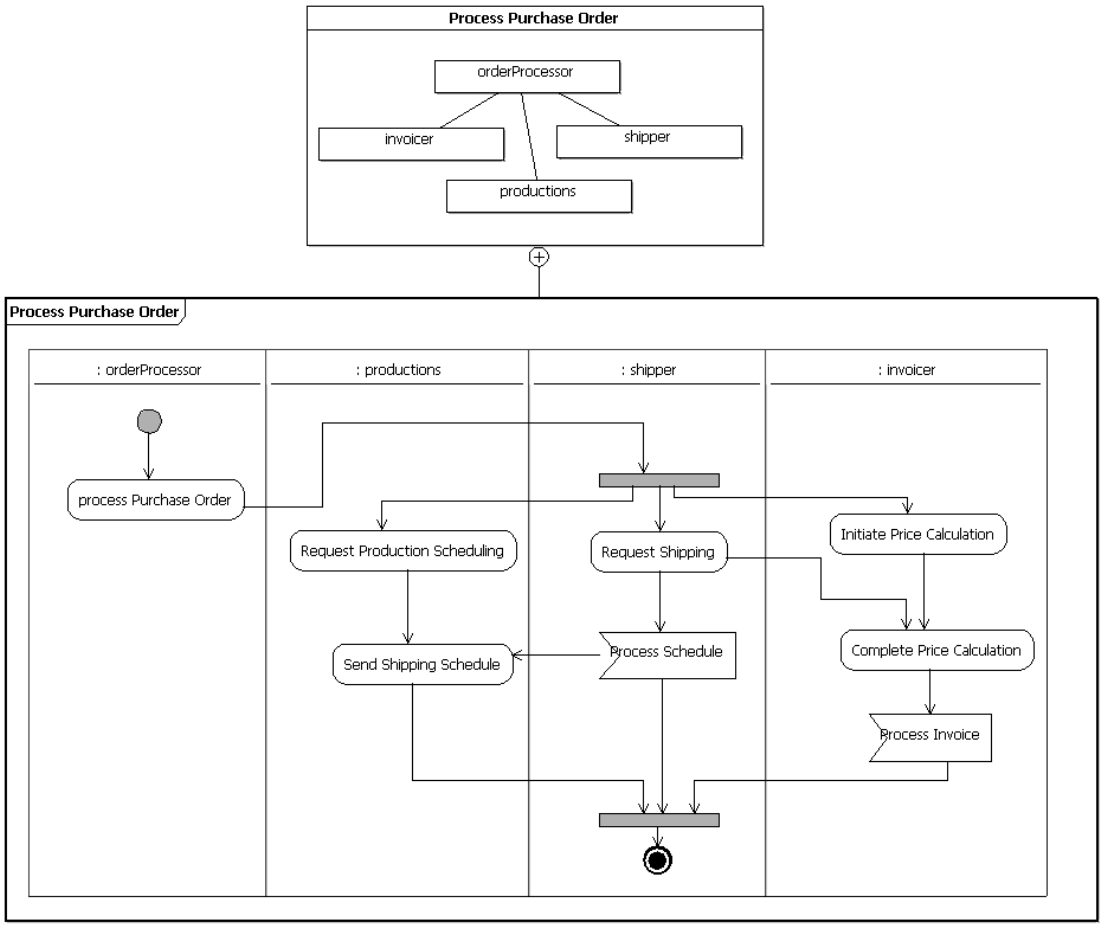

Figure 106: Process Purchase Order Contract

The Process Purchase Order collaboration in Figure 106 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 107. These Interfaces have Operations which 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.

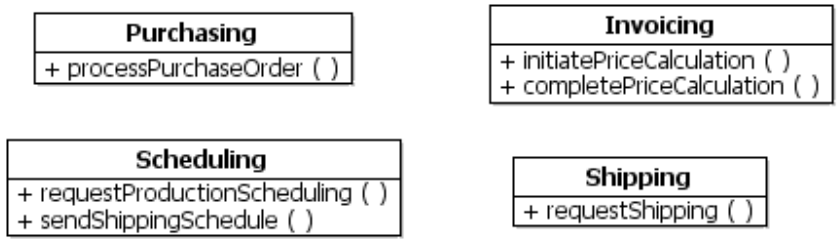

Figure 107: Interfaces Listing Role Responsibilities

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 the development process 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 108 shows a view of the service interfaces determined necessary to fulfill the requirements specified by the collaboration in Figure 106. This view simply identifies the needed service interfaces, the packages in which they are defined, and the anticipated dependencies between them.

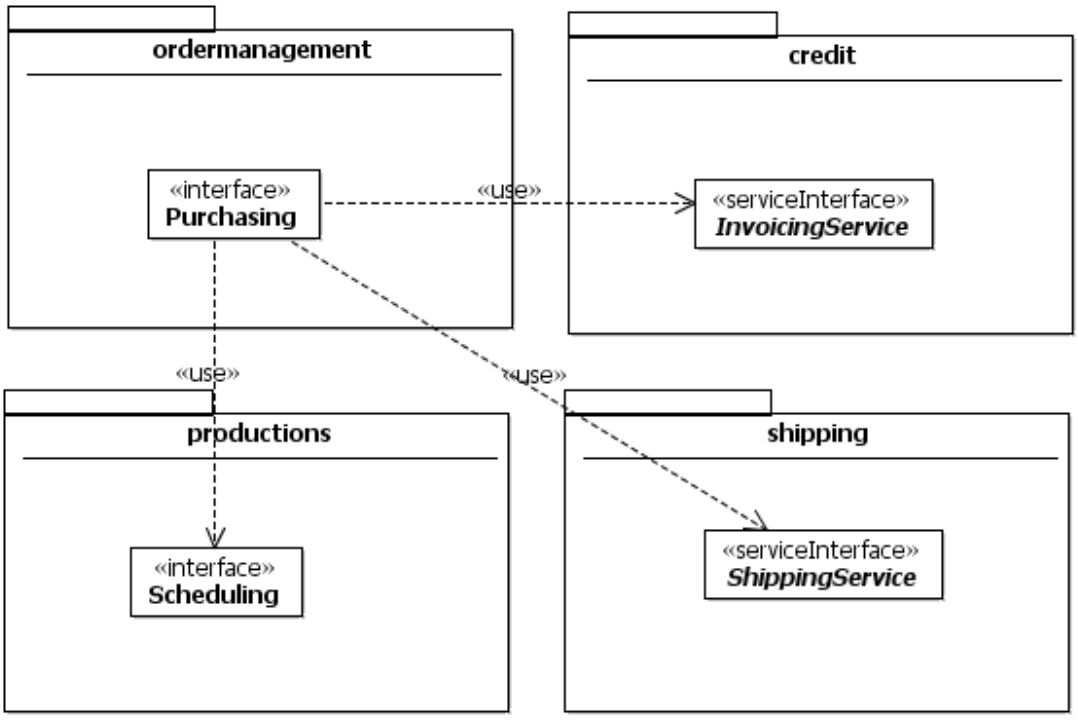

Figure 108: Identified Service Interfaces

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.

Invoicing

Figure 108 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 109 shows a ServiceInterface that defines invoicing services. This ServiceInterface provides the Invoicing Interface and requires the InvoiceProcessing Interface.

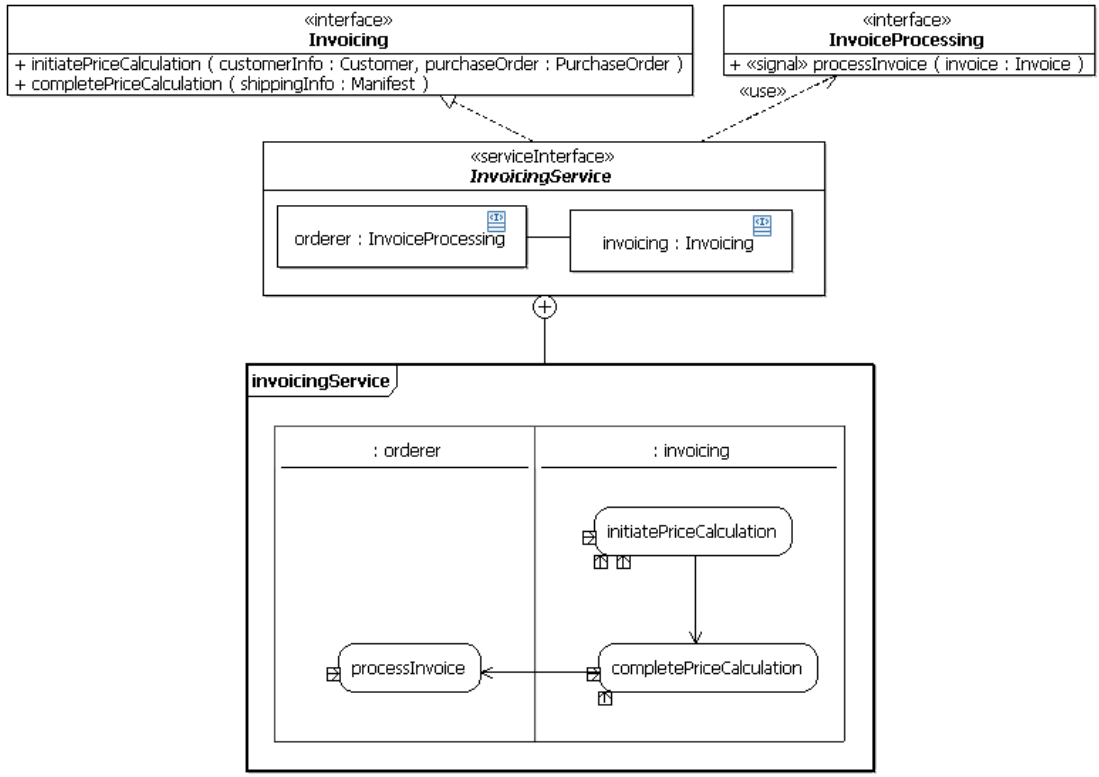

Figure 109: InvoicingService Service Interface

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 which is consistent with the behavioral part of the Process Purchase Order collaboration.

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 110: The Scheduling Service Interface

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.

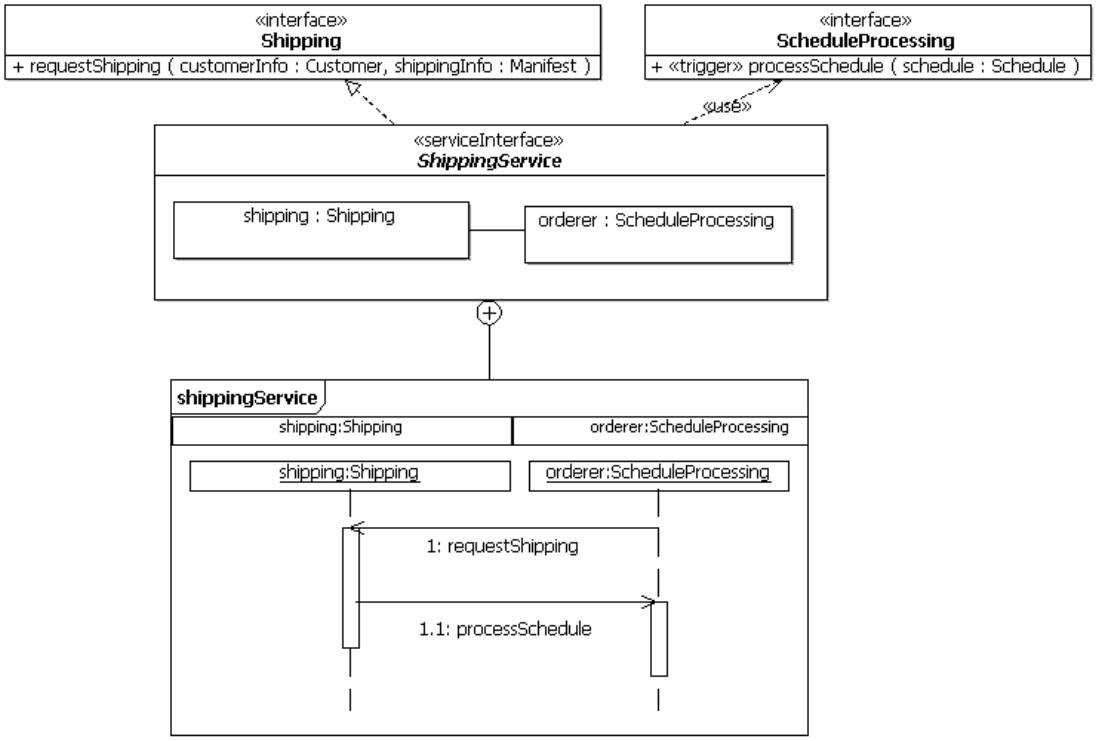

Figure 111: ShippingService Service Interface

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 112.

Figure 112: Purchasing Service Interface

Now 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 an SOA solution is to determine what participants that will provide and consume what services, and how they do so. 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 capability provided by a service participant must be implemented somehow. Each capability (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.

Invoicing

The Invoicer service Participant shown in Figure 113 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 and can also specify the possible bindings provided by the Invoicer Participant for use in connecting with other service participants.

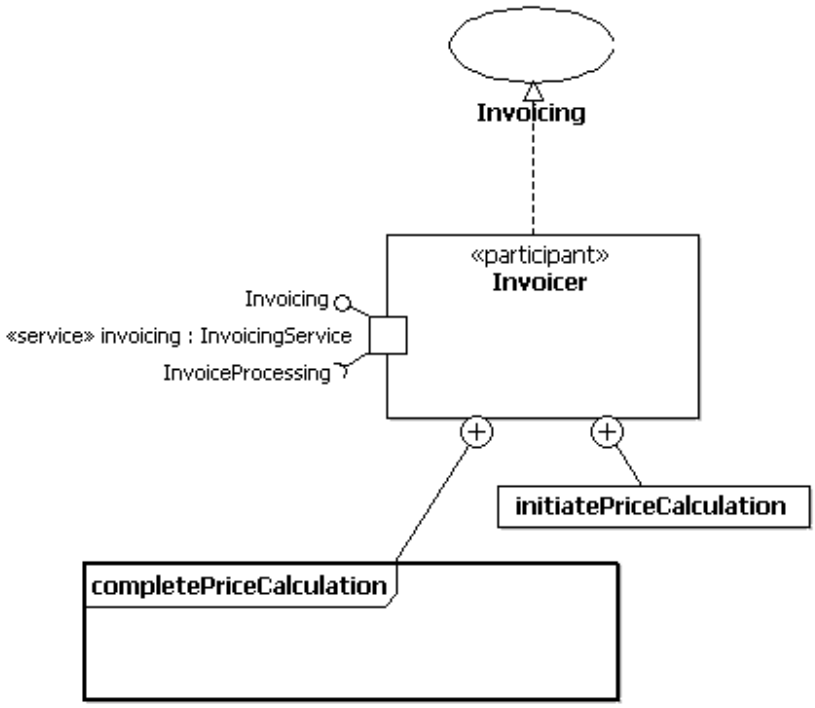

Figure 113: Invoicer Service Provider providing the invoicing service

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

Figure 113 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 shown in Figure 114. 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.

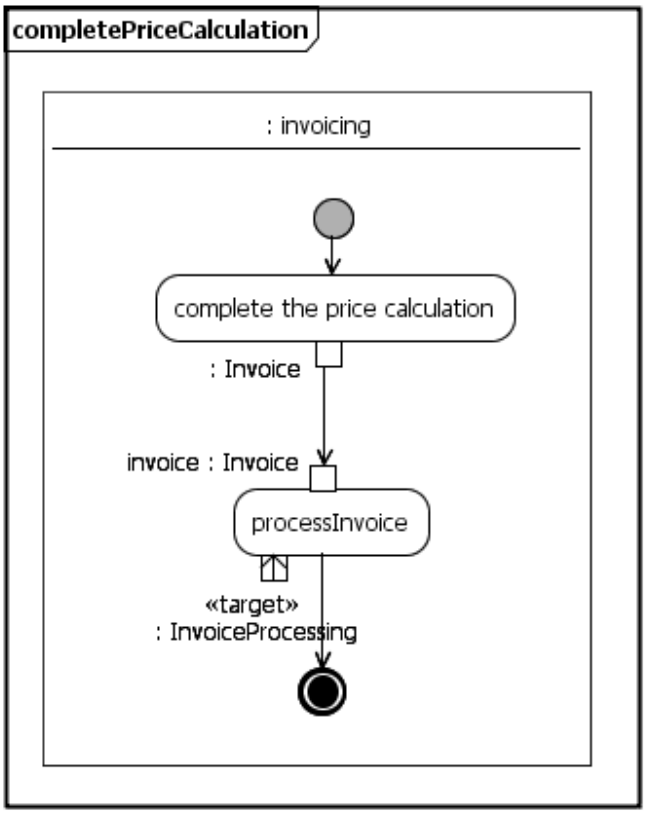

Figure 114: The Design of completePriceCalculation

Production Scheduling

The Productions component shown in Figure 115 provides a scheduling service defined by the Scheduling service interface. In this case the type of the Service simple Interface.

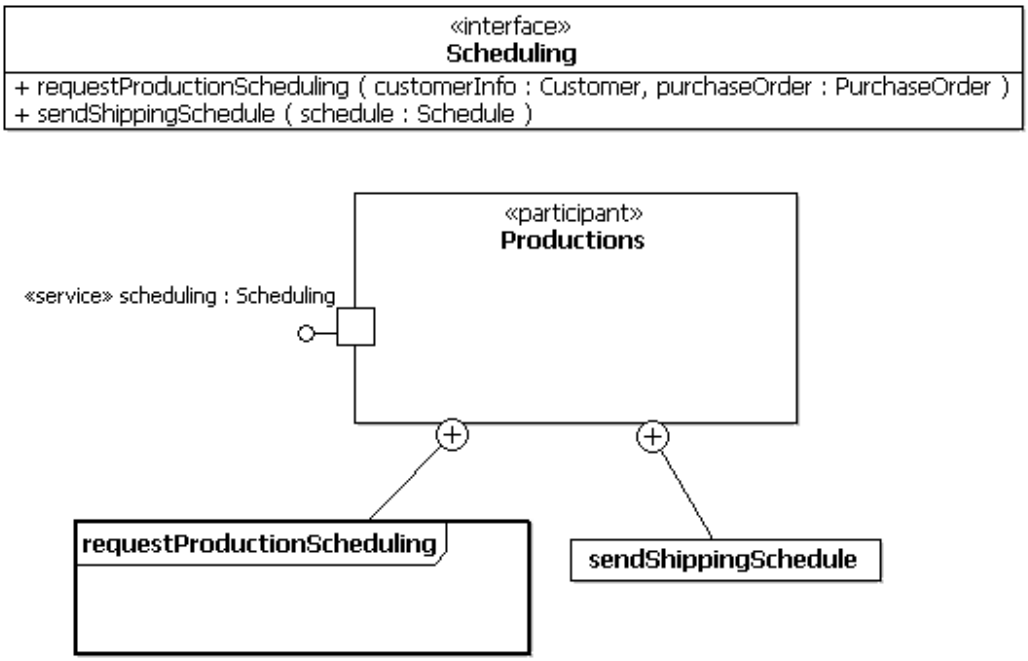

Figure 115: The Productions Service Provider

Shipping

A Shipper specification shown in Figure 116 specifies a service provider that provides a shipping service defined by the Shipping service 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.

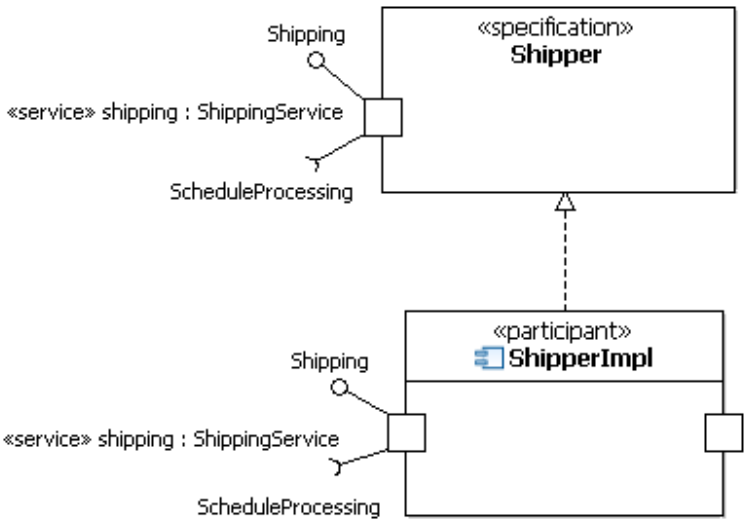

Figure 116: The Shipper Service Provider

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.

Purchasing

The purchase order processing services are specified by the Purchasing interface, and provided by the OrderProcessor provider as shown in Figure 117. This participant provides the Purchasing Service through its purchasing port.

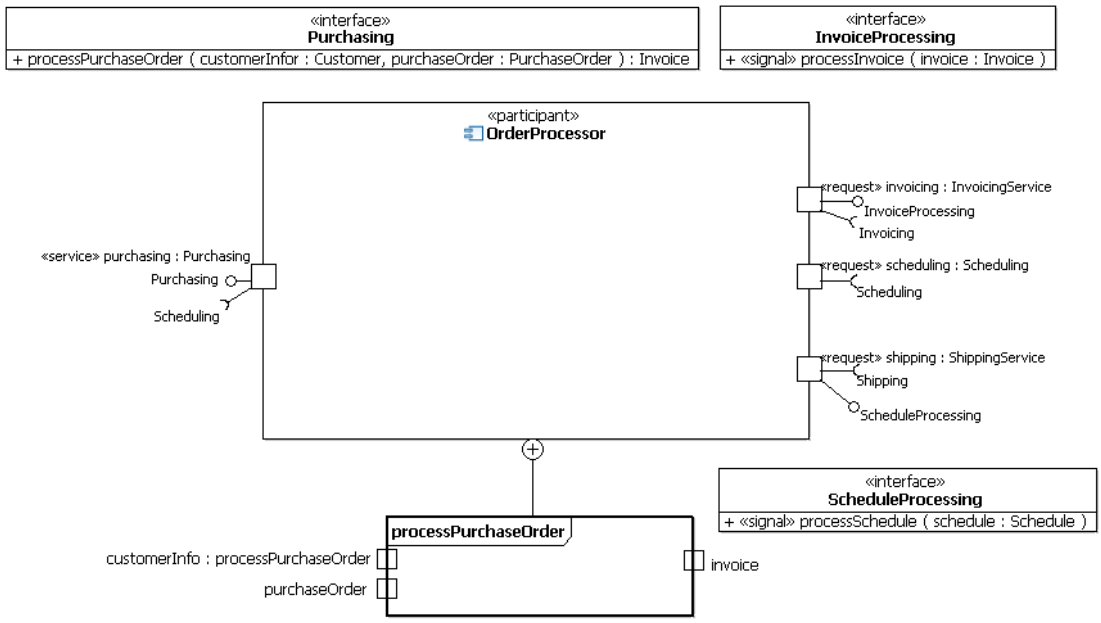

Figure 117: The OrderProcessor Service Provider

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 118.

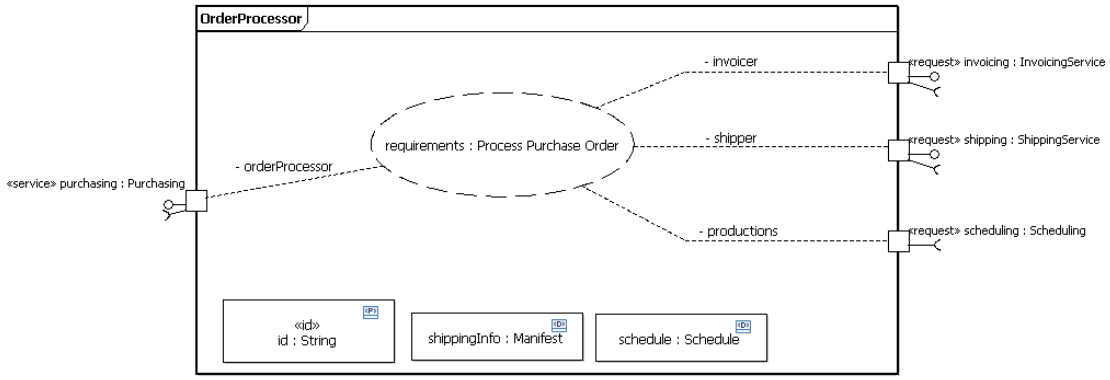

Figure 118: The Internal Structure of the OrderProcessor Service Provider

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 119. The processPurchaseOrder operation is the specification of the processPurchaseOrder Activity which is an owned behavior of OrderProcessor.

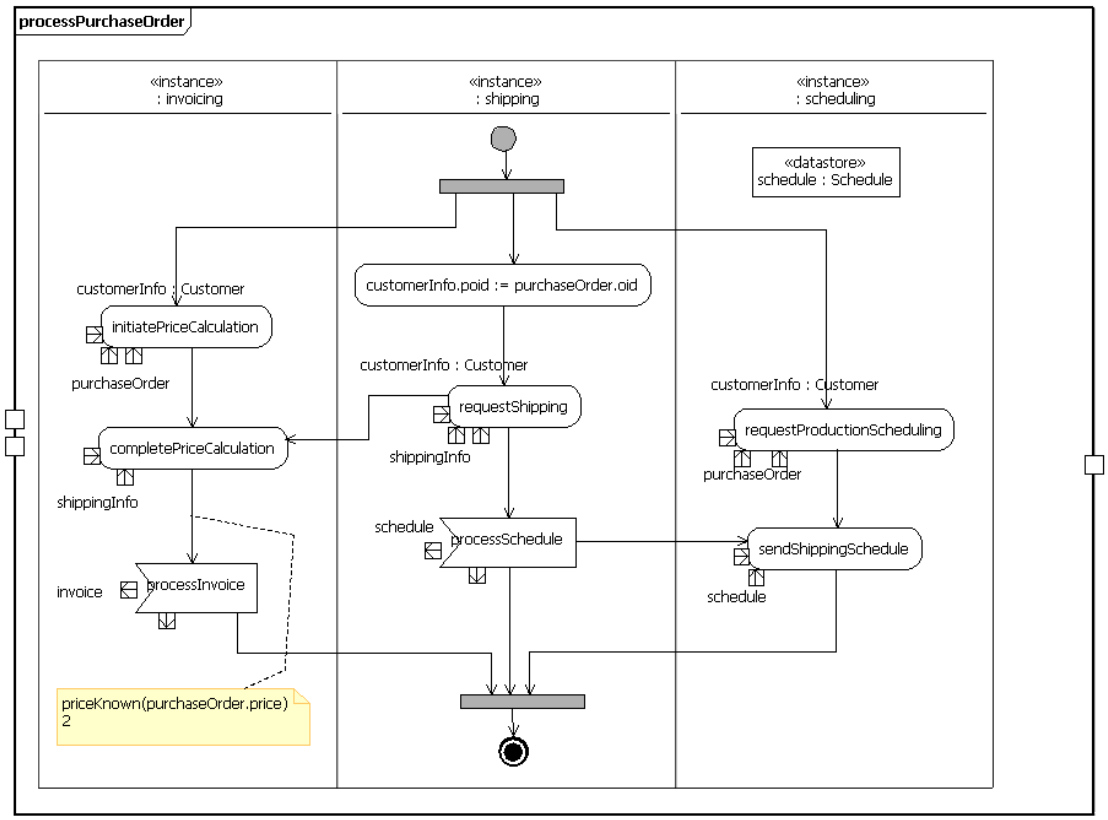

Figure 119: The processPurchaseOrder Service Operation Design

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 indicate that it fulfills the requirements specified in the collaboration shown in Figure 106. 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 106 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 118.

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, Invoider, 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 Manfacturer Participant shown in Figure 120 represents a complete component that connects the OrderProcessor service provider with other service providers that provide its required services.

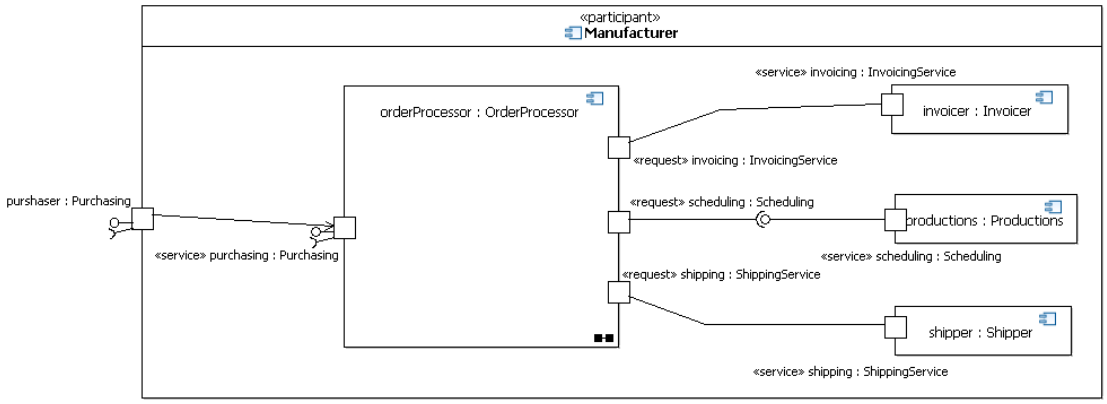

Figure 120: Assembling the Parts into a Deployable Subsystem

Figure 120 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.

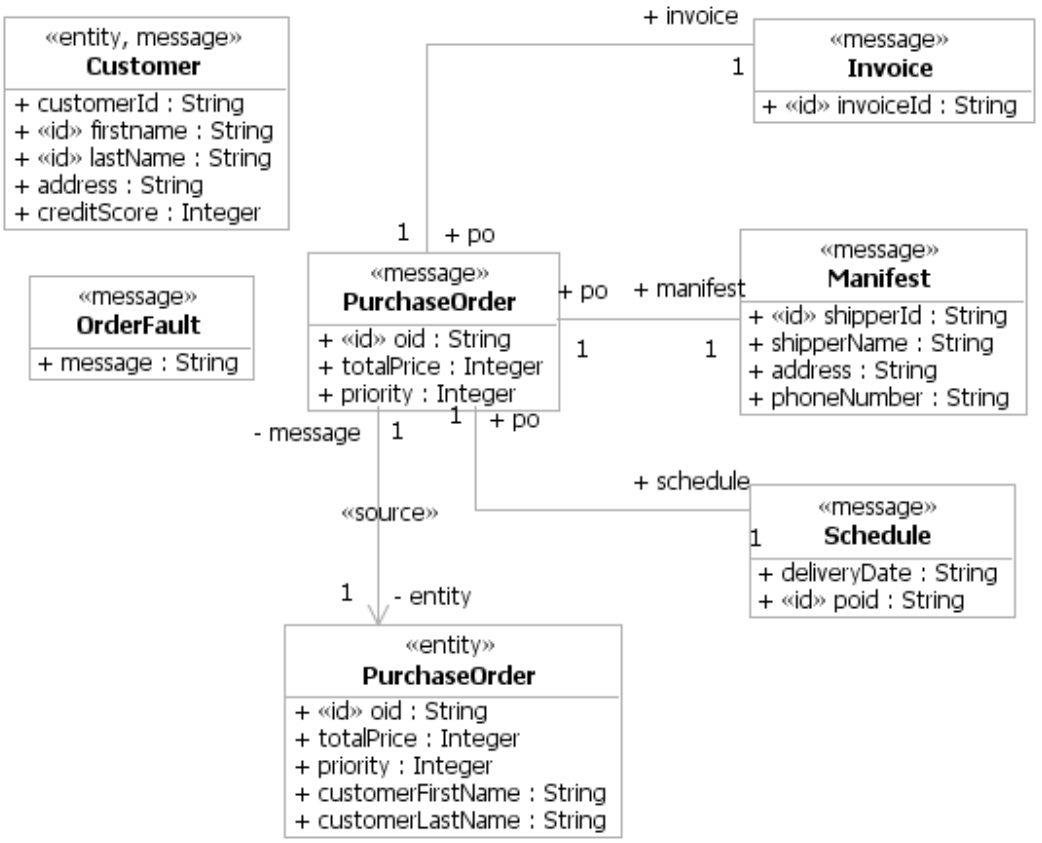

Figure 121: The CRM Data Model

Annex D: Purchase Order Example with Fujitsu SDAS/SOA

This section provides an example of a services 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 submission. It grounds the concepts in reality, and shows the relationships between the parts. The example is elaborated in other sections in order to explain submission 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.

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, stat 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.
- An 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.

Each specification model

(1) Entity Model (Class diagram)

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.

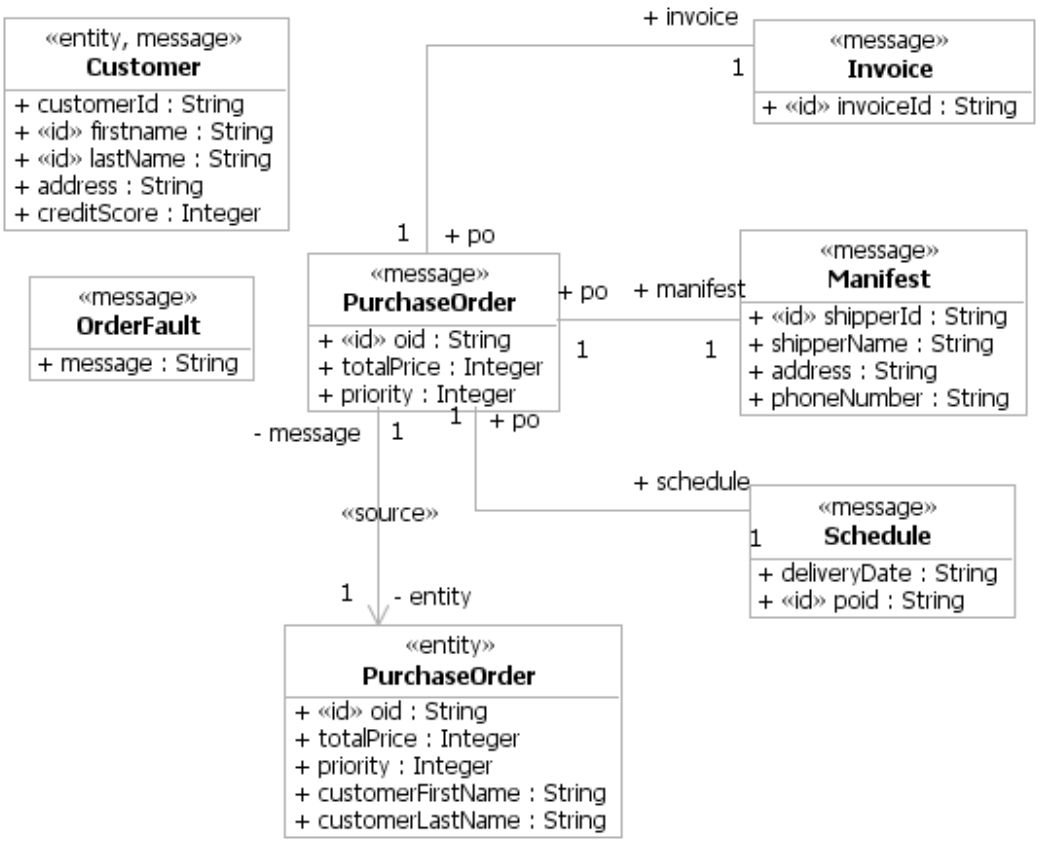

Figure 122: Class diagram

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 following state machine.

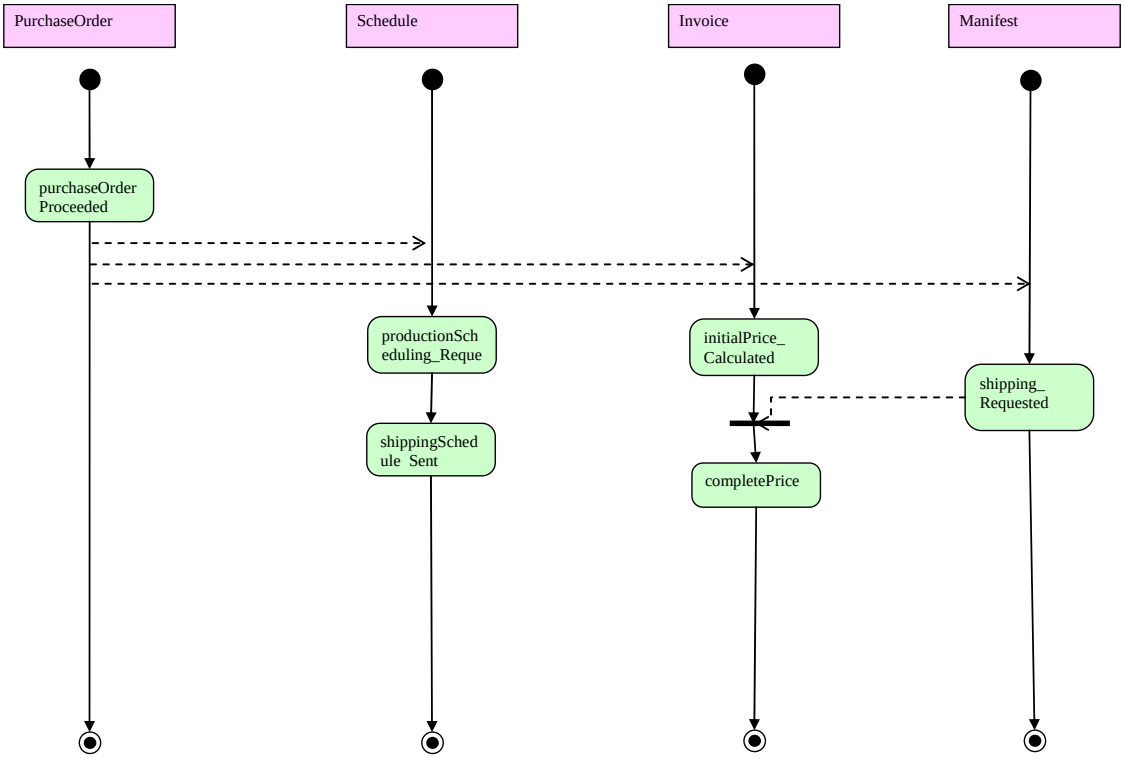

Figure 123: State Machine

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 E: SOA Patterns and Best practices

This appendix describes some interesting but non-normative SOA patterns and best practices and how they are addressed by this submission. Note that this section expresses opinions that seem to be gathering consensus as best practice but others may not agree with all of these opinions. This section is informative, not normative.

Separation of concerns

How can we hope to achieve widespread and dramatic transformation of our organizations to be more agile, customer focused and efficient? How can we deal with the complexity of our organizations, relationships and technologies in such a way that we are agile? Has the very success of our vertically and horizontally integrated organizations made it impossible to respond?

There are general and proven approaches that have succeeded over and over to manage complexity with effectiveness and agility. There have been groups, organizations, systems and projects that have overcome these hurdles to achieve to achieve results. What makes this work?

These same problems have been dealt with in manufacturing, government, defense and complex information systems. The unifying principle for success was identified by Edsger W. Dijkstra, a famous thinker in the area of software and software architecture.

That unifying concept is ***separation of concerns.***

Separation of concerns is what allows us, limited humans, to deal with the complexity and interdependencies of diverse needs and resources. It is what allows us to achieve a joint purpose without a centralized control. It is what allows us to design, build and run huge organizations and information systems. While Dyjkstra focused on software, the principles of separation of concern can be applied to any "complex system", be it government, business, military, natural or technical. The modern "network centric" approach to warfare is another expression of the same concept, as are the proven management principles of delegation and a focus on core competency.

Separation of concerns is simply the art of focusing ones attention on a particular aspect of a problem, organization, mission or system. This focusing of attention provides both a way to **comprehend that aspect**, to **understand how it relates to other aspects** and to accept that **other aspects may be under the control of others**. Our challenge is to understand how to separate our concerns while making sure that these concerns come together to achieve our mission – this is the art of architecture and design.

Delegation – business and technical

Great managers know that they can't do everything, they are great at delegation and collaboration – at assigning responsibility, letting good people achieve and making sure

all the parts work together. Great managers have learned how to separate concerns along the dimension of responsibility and action so that their team works effectively. Thus delegation is a powerful tool for separation of concerns based on delegating responsibility.

Delegation allows creativity and diversity in how responsibilities are met, while putting constraints on that creativity and diversity so that the goals of the whole organization are achieved.

Encapsulation – business and technical

Encapsulation is another term that is popular is systems architecture, but that applies just as well to organizations. Encapsulation is separating the concern of **what** vs. **how**. In computer terms this is sometimes called the "interface" vs. the "implementation". One thing the great manager is doing when they are delegating is encapsulating a responsibility – this is where the creativity and diversity come in.

A business example

If a manager is telling someone exactly what to do, this is assignment, not delegation. Assignment is a powerful tool, but it has problems in the large. Eventually the responsibility, the "what" has to be separated from the details of how something will be achieved.

Consider two scenarios. In one case a detailed process for accounts receivable has been "wired" into an organization. The details of how A/R is done are specified in great detail and it is integrated into all parts of the organization. This kind of tight integration is called for in some situations and is the basis for some management theories. In the other scenario A/R is delegated to a specific group of experts, experts who have a clear responsibility and interact with customers and other parts of the organization to gather information and assist where such information is required. What A/R does for the organization is clear – but exactly how they do it is not so wired into the organization. What if management decided to outsource accounts receivable? In the first scenario the organization would be dramatically effected, they would have to change their processes and the way they work together. In the second scenario the change could be almost invisible, because in the second scenario A/R was already "acting" like an outsourced organization – they were providing a service to the organization without being wired into it. This starts to provide a hint as to why a services approach is a good way to think about our organizations.

Now think outside your organization, isn't your customer (or whoever your organization serves) delegating some responsibility to you? Aren't you delegating responsibility to your suppliers? Externally and internally there is a need to encapsulate "the other guy" so that we can play in a larger community while not being overly concerned about how they work "inside". It should be starting to be clear that "encapsulation" is needed to structure organizations and missions and that this is the business side of SOA; SOA as "business

services" that interact and, together, achieve the goals of the organization and the communities in which it operates.

A technical example

In software the concepts of encapsulation are well established, in fact they are the basis for most quality software architectures. The "object oriented programming" revolution of the 80's and 90's is largely based on encapsulation – that "objects" have interfaces and associated required behavior, but how the objects are "implemented" is hidden. This hiding of implementation was controversial at first, because it seems that it could impact efficiency (the same comment is made when businesses encapsulate and delegate). But the complexity and fragility of systems that do not employ encapsulation has proven to be a much more substantial problem than the loss of efficiency – the agility and stability are worth the cost.

With the onset of the internet and the ability for widely distributed systems to interact, the need for encapsulation has multiplied. Our systems now routinely work with other systems inside and outside our sphere of control, it is simply unreasonable to think we can impose on their implementation or their process, we can only deal with what they expect from us and what we expect from them – as we will see, this is an essential element of SOA as a technical architecture.

If the "pattern" of SOA doesn't seem much different for the business and technical example, you have caught on – that's the key.

Community and Collaboration

A central driving theme in business, government and defense is collaboration – bringing together diverse groups, individuals and resources to achieve a shared goal. Collaboration exists within a community – some groups that wants to or needs to work together. The capability these communities need is collaboration, the ability to work together.

This drive towards communities and collaboration is so deep that it impacts the way we think about our organization, from supply chains to network centricity. The focus is now on "joint action" rather than individual actions. The environment in which we operate becomes a driving force for defining our purpose and how we fit in to the community, how we collaborate and the value we provide.

Communities and collaboration fit hand-in-hand with delegation and encapsulation – only with separation of concerns are we able to be sufficiently agile to work within a dynamic community, only by "encapsulating" our details from "the outside" can we remain efficient and agile and only by delegating responsibility to others can we help achieve the goals of the community. Helping the community achieve its goals correspondingly provides value to each participant.

This suggests a shift in thinking from one that is focused on "our process" to "how we collaborate in a community". Our process is not the center; it is a means to providing

value to communities. That value is realized by the services we provide to the community and accomplished using the services of others. These services are realized by a combination of organization, resources and technology, working together. Communities can be large or small and have almost any purpose. Anytime there is “working together”, “customers”, “collaborators” or “suppliers” there is some community, here are a few examples;

- Retail buyers and sellers
- The multinational force in Iraq
- The accounts payable department
- The U.S. Central Contractor Registration “CCR” (The vendors, manager of the information and consumers of the information)
- Internet routers
- The federal supply service
- Insurance providers
- Visa International
- The 9th Infantry Division
- Facilities Management
- The providers and consumers of internet time service (NTP)
- General Motors

Specification and Realization

The separation of specification from realization is one dimension of separation of concerns. This separation of concerns separates what is required from how it is realized but does not imply that there is any particular contract or interface – just requirements. «specification» is a UML keyword used to mark some element as being a requirement without details of how something is done.

The «specification» keyword when applied to a SoaML Participant indicates what Services and Requests are provided and required, as well as constriants on how those services must be implemented, but without providing a specific implementation. Specification Participants are used to decouple classes of consumers and providers from particular implementations, people or organizational structures. A consumer Participant could, for example, have a Request connected to a Service of a «specification» Participant. This separates the specification of a participant from many possible implementations. It is possible to substitute such a Participant with any other Participant that realizes the same specification Participant. A particular realization of a specification participant can be selected and bound during modeling, when the model is transformed into a platform specific model, when the participant is deployed, or at run time. The use of «specification» could also support people, organizations or systems that can interchangable play the same role.

A Participant can also adhere to ServicesArchtectures, provide and require services, and contain behaviors which specify the choreography for using the provided services, and how the required services are expected to be used. Any Participant must be compatible with all all the «specification» Participants it realizes through ComponentRealizations. To be compatible, the Participant must have the same or more Services and Requests, and the beahvior methods of each provided service operation must be semantically consistent with the corresponding behaviors of the realized specifications.

Collaboration and CollaborationUse and ServiceRealization are similar in that they all separate concerns. A Collaboration may specify the requirements that must be fulfilled, but does not overly constrain the architecture used to fulfill them. This is because Collaborations are more decoupled from the Participants that fulfill them through part-to-role bindings. The same part could play many roles in the same or different

Collaborations, and could have more capabilities than the responsibilities of all fulfilled roles. A ComponentRealization however is more constrained. The realizing Participant is architecturally constrained by the realized «specification» Participant.

Realization is also different then Generalization/Specialization. A specializing subclass inherits the ownedAttriabutes and ownedBehaviors of its superclass. A realizing classifier does not inherit the characteristics of its specification classifiers, rather it is bound to provide something that is consistent with those characteristics. SoaML does not specify the semantics of consistency other than the provided and required services must be compatible. 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 Participant. But how this is determined based on flow analysis is not specified.

Managing State & Sessions

State

A common design goal for distributed technology services is that they be stateless. This is often very desirable in order to limit the coupling between consumers and providers. For stateless services, a consumer does not need to connect to a particular provider, any provider will do. Subsequent invocations of service operations by the same consumer may be handled by a different provider. This reduces the work for execution environments because they do not need to correlate a consumer with a specific provider on each service request, and gives them the flexibility to pool providers for efficient resource management. A related reason for using stateless services is that there is no need to identify a specific participant in a service interaction. Distributed identity can be very difficult to manage because there is no central coordinator to generate globally unique identifiers (GUIDs) to identify the participant or to ensure a GUID is never reused.

A third reason for using stateless services is to reduce the execution environment resource load in order to provide more scaleable solutions. There is no need for the execution environment to allocate resources and preserve participant state thereby allowing the environment to support more consumers with higher quality of service.

However, in reality, many consumer/provider interactions are stateful, much as we might wish they weren't. You really, really want your bank to keep track of the state of your account – and have it the same bank you put your money into. This is especially true for services that involve a long-running conversation between a particular consumer and provider involving a specific protocol defined by a ServiceInterface and ServiceContract. There are a number of strategies for dealing with such state, the first two of which support the design goal of stateless services.

One strategy for managing state is to make the consumer be responsible for its maintenance. The services are designed so all state information needed by the service provider is contained in the service message types and exchanged on each service

invocation as needed. The provider then gets the state from the operation parameters and proceeds accordingly. This approach has the advantages of simplifying the provider implementation and supporting a larger number of consumers, but it can have significant disadvantages. The amount of state information that needs to be maintained could be large resulting in large messages exchanged between the consumer and provider that could result in poor performance. Exchanging provider state with consumers may also open security risks as the consumers may have access to data they should not see or that data may be placed on a wire through which it could be intercepted or changed. In addition, provider encapsulation is compromised and the consumer may compromise integrity by changing the state in uncontrolled and invalid ways. Of course we doub't the bank would trust the consumer to be the keeper of his account balance, so this doesn't work for many real-wold situations.

The second approach addesses these issues by having the provider be responsible for the state. In this case, the state is maintained in a persistent data source and the provider reads the data sources in order to re-establish its state on each request. The provider is still stateless from the perspective of a "running program" on an application server because it is completely responsible for maintaining its state outside that environment – in the DBMS. The only thing that needs to be exchanged between the consumer and provider is information needed to identify the state to be loaded from the DBMS. This is known as correlation data and is discussed in the next section.

The third, and most general approach is to model the state as ownedAttributes of the providing Participant, and rely on facilities of the execution environment to manage the resources necessary to maintain the state. This significantly reduces the complexity of both the consumer and provider implementations at the cost of complex resource and container management facilities in the execution environment. The execution environment would be responsible for passivating participants when resources are needed, maintaining the state in some persistent store it manages, and restoring that state then the participant is reactivated on some subsequent request.

SoaML makes no assumption about how participant state is managed. Any participant can have internal structure that includes ownedAttributes representing state, or may rely on state machines, DBMS systems, legacy applications or long running, event-driven activities to implement services. It is expected that different platform implementations and transforms may restrict what state participants may have, or how that state is handled in the platform specific model. Such platforms and transforms are outside the scope of this submission. However, the submission does capture the information necessary to support such approaches. In particular, this leads to the need to support explicit correlation between consumers and providers in order to support long-running, perhaps stateful interchanges.

Correlation

Interactions between service requestors and providers must identify the particular instances with which they are interacting. This is because there may be state information that has to be correlated in long running transactions with interaction protocols involving

multiple, related message exchanges. Typically OO systems have hidden this identification in system supported opaque identifiers based on memory addresses - self or this pointers, or system generated opaque URIs. This can be fragile and difficult to manage in a distributed environment because there is no central broker that manages the identifiers ensuring they are meaningful, stable, globally unique and never reused.

BPEL has taken the approach that the developer is responsible for providing message data to correlate requests and replies, not the system. It assumes the identifying information is somewhere embedded in the exchanged messages and that this business information is both a meaningful identifier, and stable. BPEL uses a WSDL extension to define properties for candidate identifier fragments, and property aliases to specify how the fragments get their values from exchanged messages through a query. The BPEL process defines a correlation set which is an ordered list of properties that make up a candidate identifier for instances of the process. Then each request and reply activity references a correlation set that indicates how the candidate identifier is obtained from the particular operation's input or output parameters. Different request/reply pairs for the same process may use different candidate identifiers for the process, or the same identifiers may be derived from the particular operation’s parameters using property aliases with different queries.

When a BPEL process is invoked, the runtime uses information gathered from the input message to identify an existing or newly created instance of the process. This information is based on the queries specified for the property aliases of the properties in the correlation set. Then when the process replies back to the client, the same correlation set is used which results in the same values of the identification properties being sent back to the client. On the next interchange the client invokes another service operation provided by the same process using the same correlation data in the message. It is the client’s responsibility to ensure the correlation data is placed in the messages in the right place, and has the right values in order to communicate with the correct process instance, and that this correlation data remains consistent throughout interactions with the same process instance.

Since the BPEL runtime platform expects the developer to manage request/provider correlation using business data, we have to expose this to the modeler in UML too. Ideally these identifiers would be factored into separate data types (often primitive) so they can be easily distinguished and reused in a uniform manner, similar to this pointers in C++ or self in Smalltalk and UML. However this is generally not practical as the identifying information is often embedded in multiple parameters, and may be different for different calls to the same process instance. So we need a more flexible way of specifying candidate identifiers, and how they are derived from data in the parameters of invoked operations.

The SoaML provides a simple facility that may be used to enable two ways of managing correlation, implicit and explicit using «id» properties.

Implicit Correlation

When using implicit correlation, transforms will have to automatically generate everything required to correlate requests and responses. Each Participant could have an «id» property. For implicit correlation, the type of the «id» property can be either a UML2 PrimitiveType or DataType (that has no identity of its own) in order to support composite id's. However, the Participant would have only one «id» attribute. This id attribute must be public. In the generated interfaces for the corresponding BPEL process, this attribute would be included as the first input parameter to the operation. The «id» property of the Participant is used to generate a corresponding correlationSet in the BPEL process and its associated property and propertyAlias elements in the process's WSDL file.

Explicit Correlation
An «id» property is still used to identify instances of an activity or component. However, the type of an «id» property modeling explicit correlation would be for example an «identifier» class (not part of SoaML). The id properties would be typed by an identifier class.

The properties of an identifier class are the properties that taken together identify an instance of the component - a candidate key for the component. This is the same as for implicit id properties. For distributed systems, the actual identification data is part of the business data in the messages exchanged between requestors and providers, i.e., the parameters of the invoked operations. What we need is a way of specifying how the in parameters of each operation provided by a participant provide data to the properties of its identifier.

The ownedAttributes of an «identifier» class could be all derived. How they are derived could be different for each operation provided by the component because the identification data could be embedded in different parameters.

An «identifier» class could have a «correlation» directed association to the Interface containing the operation whose parameters contain the actual correlation data. Each property of the «identifier» class is derived from the parameters of operations in its associated Interface. The derivation of the property is specified by a constraint attached to the class.

A component could have many id properties that either represents different ways to identify the same component instance, or different ways of accessing the identity information. As a result, a component or activity using explicit correlation may have many «id» properties each with different «identifier» type. These types can have different derived properties, or can be subclasses of other «identifier» types that use different constraints to get the same identification information from different parameters.

SOA Interaction Paradigms

Under the wrappers of various SOA distributed computing technologies there has emerged a clearer distinction between some basic paradigms and approaches that have, in

fact, existed for some time. These are the RPC, Document Centric Messaging and publish/subscribe paradigms. These distinctions are important because they present true architectural choices that can affect everything from the planning to deployment of an information system or business process. The systems architect today has a set of choices and supporting technologies for each approach. SOAP. WSDL and web services currently embrace both the RPC and document messaging paradigms. SoaML embraces all three distributed paradigms so the architect can choose the most appropriate.

Remote Procedure Call (RPC)

RPC has a history dating back to the early days of distributed computing and was brought into the object world with OMG CORBA, Microsoft DCOM, Java EJB (Entity and session beans) and SOAP RPC. The basic tenant of RPC is to make the distribution layer transparent to the programming language by making procedure calls flow through some kind of technology layer that handles distribution. All of these RPC mechanisms are object oriented (there are non object alternatives as well, but no longer mainstream) and can map directly to language interfaces. SOAP-RPC is the web-services version of RPC based on SOAP message and XML. RPC systems tend to have many "fine grain" operations that take very small parameters – frequently just primitive data types.

Since RPC is based on the method call, RPC is inherently synchronous (There are some asynchronous ways to use SOAP RPC or CORBA – but this is rarely used) and is designed to handle the data types typically found in programming languages. A remote interface with methods and method arguments is the basis for defining interoperability. Since most modern RPCs are object oriented, this is also known as distributed objects. SOAP RPC differs from classic distributed objects in that there are no object references or ways to create distributed objects in SOAP message – all the "distributed objects" are assumed to be pre-defined. Generally these distributed objects are value objects or Data Transfer Objects (DTOs) that do not have identity, are exchanged by value (or copy), and can be freely exchanged from one address space to another.

Think of RPC as like making a phone call with an automated attendant – you are "on the phone" and waiting for each response.

Document Centric Messaging

The document centric messaging paradigm assumes a "document" as the basic unit of interaction, perhaps with attachments. Documents richer data structures that are sent between distributed systems, frequently to a static logical address or via an ESB. Document messaging may be connected or based on message queues for later delivery. Connected messaging may also be synchronous or asynchronous. Document messaging frequently assumes that the conversation between parties is long-lived and persistent (e.g., A business conversation). Document centric systems tend to use medium to large grain messages that mimic the scale of typical business "forms".

Document messaging relies on some way to specify the documents used for interaction and XML has assumed premier status in this space. SoaML also provides a way to

specify what documents may be accepted at specific service locations, essentially the interface of the messaging service. SoaML may also specify document interaction ordering using UML behaviors. With messaging there is no knowledge of methods or objects on the "other side" of a conversation, the binding between the parties is limited to the documents exchanged. It is expected that the "other side" knows what "method" or "application" to call for a particular type of document – or will reject it.

Think of document messaging as being like sending a message to someone's inbox.

Publish/Subscribe

Both RPC and document messaging assume a conversation between specific parties, which is frequently required (When you buy something you want to know the supplier!). However it is sometimes unnecessary to have such directed conversations. Where there is just the need for one system (or person) to react to "events" within the enterprise (or within another enterprise) then publish-subscribe is used. Publish/ subscribe is similar to document messaging, but an intermediate messaging broker is used. Notice of interesting events are "published" to this broker and anyone with sufficient rights can "subscribe" to any event notification. The producers and consumers of event notifications do not even need to know about each other, they just need to know about the event and the service broker. As a result, the coupling between consuming and providing participants is reduced as the connection is made at the latest possible time, is very flexible, and can easily change while the system is running.

Publish/Subscribe is always asynchronous and frequently built on a messaging system backbone. Product names in this space are: MQ-Series, JMS, MSMQ and Sonic-MQ.

Think of pub/sub as being like posting a note on a bulletin board – or a radio broadcast.

Criteria for picking the paradigm

Loose Coupling

Loose coupling means that the systems communicating can be implemented, evolve and be managed independently with minimal *required* common infrastructure. Loose coupling is absolutely required between independent business partners and between systems of different management domains. Unnecessary coupling between systems makes them hard to implement and very hard to evolve and change over time (and everything changes over time). Much of the problems with legacy systems results from tight coupling, so loose coupling has been recognized as a good architectural principle even within one application – because even one application needs to have "parts" that are build and evolved independently.

High Cohesion

High Cohesion is the degree to which independent systems work together to further business goals – the "contract" between systems working together. The higher the

cohesion the less chance there will be for inconsistencies to develop or for the underlying process to break because the supporting systems are not sufficiently aligned.
So cohesion, which is a business goal is a necessary and good thing, while tight coupling should be avoided.

Performance

Another important factor is performance. It should be noted that the performance difference between these paradigms is almost entirely subject to the supporting technology and the required resilience of the communication path. For example, a highly secure and reliable RPC system will be slower than a “best efforts” pub/sub. It is popularly held that RPC is “faster” than the other solutions, but this has not proved to be a reliable indicator, often because the granularity of the operations is not properly managed to take distribution into account. Multiple communications, buffering messages to disk, resource locking, security and guaranteed delivery are the costly coin of distributed computing.

In a large scale environment the effective use of resources must also be considered. Synchronous solutions will have faster turn-around time for a single message but will also place more demand on the infrastructure and may not scale as well – the system as a whole may be slower. Asynchronous solutions must be very well engineered to perform in an interactive systems environment – such as a user interface. Synchronous solutions must be very well engineered to scale to many users.

All distributed technologies have substantial overhead. Just the cost of “marshalling” data into some form of middleware protocol (be it XML or CORBA-IIOP) and sending that data over a network to another process is quite high and unavoidable. Each time a message is sent there is overhead as well as each time a new connection must be made. Added messaging infrastructure like a disk-based queue for guarantied delivery and security adds to the overhead.

Due to the overhead of each message, distributed systems should use “large grain interactions”, putting as much information in each message as possible, even if some of that information is not needed every time.

Where a real-time response is required a synchronous protocol may be preferred since it can send one document or method and get back a response in one connection. However, the additional system load of all those connections sometimes makes synchronous systems impractical for large numbers of users. This is an area of some debate and ongoing research.

Granularity

As discussed under performance – large granularity (big messages) is a substantial indicator of performance. Large grain interactions – sending as much data at one time as you can, is often preferred because the cost of each round-trip distributed interchange is often much higher then the cost of copying larger amounts of data in a single interchange. Older EDI solutions were large grain, but lacked the flexibility of XML today so they

become coupled. Solutions need some easy way to evolve messages without breaking existing systems.

There is some debate over whether large grain or fine grain messaging is better for supporting loosely coupling and high cohesion, carful attention to architectural principles is required in either case. Large grain, loosely coupled interactions will encapsulate a unit of business functionality using the document centric approach. Of course messages need to be "right sized" so that they fulfill a business need without unnecessarily coupling together unrelated data or processes. Systems for processing or using larger grain messages should also be built to be resilient to changes in those structures with zero or minimal impact on the implementation. Interfaces with messages that are overly fine grain will "implicitly couple" those systems because so many messages have to be sent in just the right way to achieve a business result. Designing a system for high cohesion and loose coupling requires a good architecture, but pays back in reduced maintenance costs and greater agility in the long run.

An approach that is gathering some recognition as an emerging best practice is the use of Ontologies and the semantic web to define loosely coupled documents, since that approach is less sensitive to structural and terminological differences and changes between document definitions. The use of Ontologies and the semantic web to define SOA messages is fully consistent with the SoaML approach.

Distributed Transactions

From a design point of view it would be nice if we could make a large distributed system behave like a local system, but this is not always possible. In particular, distributed transactions have proved expensive to support and difficult to implement. It has also been recognized that having entire complex systems "revert" to some prior state due to something going wrong is not something found in natural or human interactions. Due to the implementation and conceptual issues it has become accepted best practice to make distributed interactions as discrete and atomic technical transactions wherever possible – that is transactions do not extend across multiple interactions and therefore there is no need for a distributed transaction. Compensating transactions are used to maintain a consistent system state when "bad things happen". Within a single application distributed transactions may be unavoidable and will require supporting infrastructure.

Blocking

Blocking is the practice of having one resource wait while another does it's work. If that other resource is a human, blocking can consume a lot of resources. But even in the case of system-system interaction, a non-blocking solution will be faster from an overall systems perspective even if a single message may take longer. Blocking is caused by the use of synchronous interfaces and single threaded applications.

Ease of development and maintenance

Ease of development is not an absolute – but depends on what you are starting with and what you are trying to accomplish. If you are starting with an existing object interface (that is sufficiently granular), RPC may be easiest. If you are starting with an EDI system

or manual document system a document orientation may be easiest. If you are integrating legacy systems, or having very loose coupling, pub/sub may be the shortest course.

Besides the ease of development the total life-cycle should be considered, life cycle concerns will be addressed by loose coupling. Systems that don't change very often may be able to tolerate a higher degree of coupling and be easier to implement.
From a top-down business perspective the document centric messaging and pub/sub solutions seem to closely mirror the typical business model. The process of designing good documents for systems to interact makes sense to business people and can even be implemented without a computer system.

In programming - the implementation an RPC system looks like method calls, so the knowledge transfer is very high. Document orientation can use generic toolkits (Like DOM or SAX) to parse and manage documents but these APIs can be clumsy. You can also convert documents into language objects with tools provided with by most systems. So if good tools are not provided, document oriented solutions may be a little harder to program. With good tools document oriented solutions are frequently faster to develop. From a maintenance point of view, less coupling means easier maintenance.

Best Practices Mindset

There is a subtle human difference between RPC and document messaging. Great architects of RPC systems have been making flexible and large grain interfaces for years, and these can perform and evolve very well. However, the mindset promoted by an RPC is that of a method – which tends to be small and very fine grain – primitive data types as parameters to lots of methods. It is very hard to change these interfaces over time – doing so breaks both sides. On the other hand – document messaging focuses the attention on creating a few large robust documents. XML provides almost "automatic" extension capability in the document paradigm.

So while you can define flexible large grain interactions with an RPC and you can make fine-grain documents – the XML messaging and pub/sub paradigms encourage and support best practices better than RPC.

Making a choice

Pub/Sub

Where there are systems that may have "side effects" from other systems (or business units with side effects from other business units) and there is not a requirement for a long-lived conversation – publish/subscribe is the best choice in that it is very loosely coupled and performs well since there is no "blocking" of synchronous calls. The design of these systems is then centered on the event types (or signals) and subscription rules. Since Pub/Sub does require a shared pub/sub domain, it may not be appropriate for B2B interactions – it works very well inside a single enterprise. However, if an interaction is or may be B2B one of the other choices may be better.

Note: It is possible to use pub/sub between independent business partners with a close working relationship. Sometimes automated EMAIL is used for this purpose.

Document Messaging

Where systems may be interdependent – requiring either a request/reply or long lived conversations, messaging may be the best choice. Since messaging may be either synchronous or asynchronous, the same application can be built to use either communication style and configured for best performance.

Document messaging works well in support of B2B, EAI, EDI and user interfaces interacting with back-end services, especially in situations where there are shared authorities or standards defining the documents.

Use of XML is very effective for messaging (and pub/sub) since it provides mechanisms for document extension, an application need only be concerned with a portion of a document and may ignore other parts – allowing the document and the end systems to evolve more independently. The process of designing effective documents aids in the understanding and loose coupling of the entire system. Since XML can be used for both messaging and pub/sub, the design and support infrastructure can be shared between the two.

RPC

If you are starting with DCOM, CORBA or EJB (Entity or session bean) you already have an RPC and converting it to SOAP may be automatic.

If you are starting with an existing object or legacy interface it may be most effective to directly expose that interface using RPC. There are good tools for making object interfaces into remote services. However, be aware that an object interface designed for local interaction may not work well as a distributed interface – they should still be large grain unless the systems are all on one small LAN.

If you are building an application and want to stay within the paradigm of that language, RPC will be easier to develop as it does not require designing documents or using a different invocation paradigm.

In general the advice of the SoaML team is to avoid use of RPC unless legacy concerns make it mandatory. The tighter coupling of RPC systems will almost always be a hindrance as the system evolves. Synchronous document messaging is just as fast as RPC and much less coupled. When faced with an RPC or API interface, it is best to combine multiple method calls together to produce a large grain document.

SOAP Messaging and RPC

The SOAP protocol supports both document messaging and RPC. The basic SOAP header provides for messaging – getting a document from one place to another. Layered on top of this is a way to represent programming language data types in a document

instance and a way to bind to methods with signatures. So SOAP RPC is basically two SOAP messages used and combined in a particular way.

Note that when using SOAP-RPC the "object" you get out of one side will have the same data as the object on the other side, but may have a very different interface – so care must be taken to not assume to much about the interface in the implementation.

SOAP RPC is designed to be driven from a programming language interface and basically makes a document for each method, with the method arguments being elements of that document. In SOAP message RPC you do not design XML, you design object interfaces that are mapped to XML. You do not use SOAP RPC to send "generic" XML documents – you use it to export object interfaces.

Since SOAP RPC uses tools to make SOAP messages it can't be faster than messages as some people believe. SOAP messaging and SOAP RPC over the same transport will perform about the same. You may want to use SOAP messaging over a synchronous protocol (like HTTP) if you want fast turn-around. SOAP messaging over a queuing infrastructure will have a higher overhead than HTTP.

If you would prefer to use typed "language objects" to interface with XML rather than generic interfaces there is some help. In Java you can use "JAX-B" to convert a document to Java objects (lots of them) or deal directly with a document API. In Microsoft you can use BizTalk for the same purpose. In both cases you will get language objects based on a DTD or XML-Schema input. One call converts the XML into these language objects. Recent advances in message handling have made it easier to have a document "pipeline" all the way through your system, without conversion into and out of fine-grain language objects.

These factors, combined with the best practices "mindset" make SOAP messaging our first choice. Evolving standards will also provide for Pub/Sub over SOAP messaging.

Summary of paradigm features

	RPC	Document Messaging	Publish/ Subscribe
Synchronous?	Usually	Sometimes	Never
Sender and recipient connected?	Yes	Yes – perhaps transiently	No, or indirectly thought exchanged events.
Long-lived conversations	Not Usually	Frequently	Sometimes
Distributed Transactions	Frequently	Not Usually	Never
Turn-around time	Fast	Synchronous – Fast Asynchronous - Medium	Slower due to intermediary – usually disk buffered
Coupling	High – data types and	Low – requires	Very Low – requires

	multiple methods must match between parties	known party to party message only	only shared pub/sub domain & data format
Specification	Object Interface	Document & Operation	Event type
Required Infrastructure	Application Server	Communications path and/or Application Server	Pub/Sub engine
Recommended for	Legacy, real time	B2B, collaboration, integration and User Interface	Collaboration & Integration
Use in SoaML	Define an operation with multiple parameters in the interface of a ServiceInterface.	Define an operation with one parameter in the interface of a ServiceInterface, or, define a signal to be received by an interface using MessageType	Use UML2 Reception, Signal, SendEventAction, AcceptEventAction.

PART III

Changes to Adopted OMG Specifications

There are no changes to any of the referenced OMG specifications. This submission only defines new capabilities that are added to UML2. These additions require no changes to existing UML2 notation, model elements, or semantics. Instead the add notation, model elements and semantics that extend existing UML2 capabilities for services modeling.